

Amiga Developers Conference Notes

Orlando, 1993

Copyright

Copyright © 1993 Commodore-Amiga, Inc. All rights reserved. The materials used herein have been reproduced with the permission of the authors indicated. Use or reproduction of such materials shall be in accordance with the authors' instructions. Commodore and Amiga are registered trademarks of Commodore Electronics Ltd. and Commodore-Amiga, Inc. respectively. This document may also contain reference to other trademarks and registered trademarks for the various products listed which are believed to belong to the sources associated therewith.

Warning

The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability).

Disclaimer

This information is provided "as is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, indirect, incidental, special or consequential, resulting from any claim arising out of the information presented herein, even if it has been advised of the possibility of such damages.

3

3

3

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks.

Amiga is a registered trademark of Commodore-Amiga, Inc.

Amiga 600, Amiga 1200, Amiga 4000, AmigaDOS, Amiga Workbench, and Amiga Kickstart are trademarks of Commodore-Amiga, Inc.

Commodore Dynamic Total Vision and CDTV are registered trademarks of Commodore Electronics Limited and Commodore.

AmigaVision is a registered trademark of Commodore Electronics Limited and Commodore.

Amiga/NFS, CBM, Commodore, the Commodore logo, and AUTOCONFIG are registered trademarks of Commodore Electronics Limited.

dBase is a registered trademark of Ashton-Tate Corporation.

68000, 68020, 68030, 68040 and Motorola are trademarks of Motorola, Inc.

Aztec C and Manx are trademarks of Manx Software Systems.

SAS and Lattice are registered trademarks of SAS Institute, Inc.

UNIX is a registered trademark of AT&T.

Novell and Netware are registered trademarks of Novell, Inc.

IBM is a trademark of International Business Machines, Inc.

Macintosh is a trademark licensed to Apple Computer, Inc.
Apple is a trademark of Apple Computer, Inc.

MS-DOS is a trademark of Microsoft Corporation.

Table of Contents

**International Amiga
Developer's Conference**
*Orlando, Florida
January, 1993*

System Software

- | | |
|--|--|
| 1. Introduction to Amiga Programming | <i>Introduction to Amiga Programmin</i> |
| 2. Compatibility from 2.0 to 3.0 and Beyond | <i>V39 and AA Compatibility</i> |
| 3. ARexx | <i>ARexx</i> |
| 4. 3.0 Graphics | <i>Overview of V39 Graphics</i> |
| 5. 3.0 Intuition | <i>Intuition 3.0</i> |
| 6. Datatypes | <i>Datatypes</i> |
| 7. Changes in the V39 OS | <i>Other 3.0 Features: GadTools, ASL</i> |
| 8. IFF | <i>IFF</i> |
| 9. AmigaGuide | <i>3.0 AmigaGuide</i> |

Hardware

- | | |
|--|---|
| 10. 68040/68060 | <i>68060: The Fourth Generation in a</i> |
| 11. 68040+ and Other Advanced Exec Issues | <i>Advanced Exec and CPU Issues</i> |
| 12. AAA Hardware and Software Issues | <i>An Overview of the Advanced Amig
Architecture and Other Future I</i> |
| 13. Low-end Hardware | <i>A1200 Hardware Developer Notes
A600/A1200 PCMCIA Slot</i> |
| 14. AA Machines, Expansion Boards | <i>A4000/A3000 Hardware Developer</i> |

Japan

- | | |
|---|--|
| 15. Japanese Localization Issues | <i>Localizing Software for Japan</i> |
| 16. Amiga Application Distribution in Japan | <i>Amiga Application Distribution in Japan</i> |

Future Directions

- | | |
|-------------------------------------|---|
| 17. Amiga User Interface Directions | <i>Upcoming Amiga User Interface
Developments</i> |
| 18. RTG | <i>Retargetable Graphics Specification</i> |

Multimedia

- | | |
|-------------------------------|---------------------------------|
| 19. CAMD and Realtime Library | <i>Introduction to CAMD</i> |
| 20. DSP | <i>DSP on the Amiga</i> |
| 21. Media Links | <i>MediaDevices</i> |
| 22. MPEG | <i>MPEG and the Amiga/CDTV</i> |
| 23. AmigaVision Professional | <i>AmigaVision Professional</i> |

Networks

- | | |
|---------------------------------|---|
| 24. AS225 and SANA II | <i>AS225 and SANA II</i> |
| 25. Envoy Overview | <i>Envoy</i> |
| 26. Neural Nets and Fuzzy Logic | <i>An Introduction to Fuzzy Logic and
Neural Networks</i> |

Product Development

- | | |
|---------------------------------|---|
| 27. AppShell and AppBuilder | <i>Writing Applications with AppShell</i> |
| 28. Object Oriented Programming | <i>Object Oriented Programming</i> |
| 29. Product Testing | <i>Product Testing</i> |
| 30. Debugging Software | <i>Using Amiga Debugging Tools</i> |
| 31. OS-Friendly Games | <i>OS-Friendly Games</i> |
| 32. CDTV Spooly | <i>CDTV Programming and Spooly Device</i> |
| 33. CD_ROM Tools | <i>ISO-CD and OPTCD & SIMCD</i> |

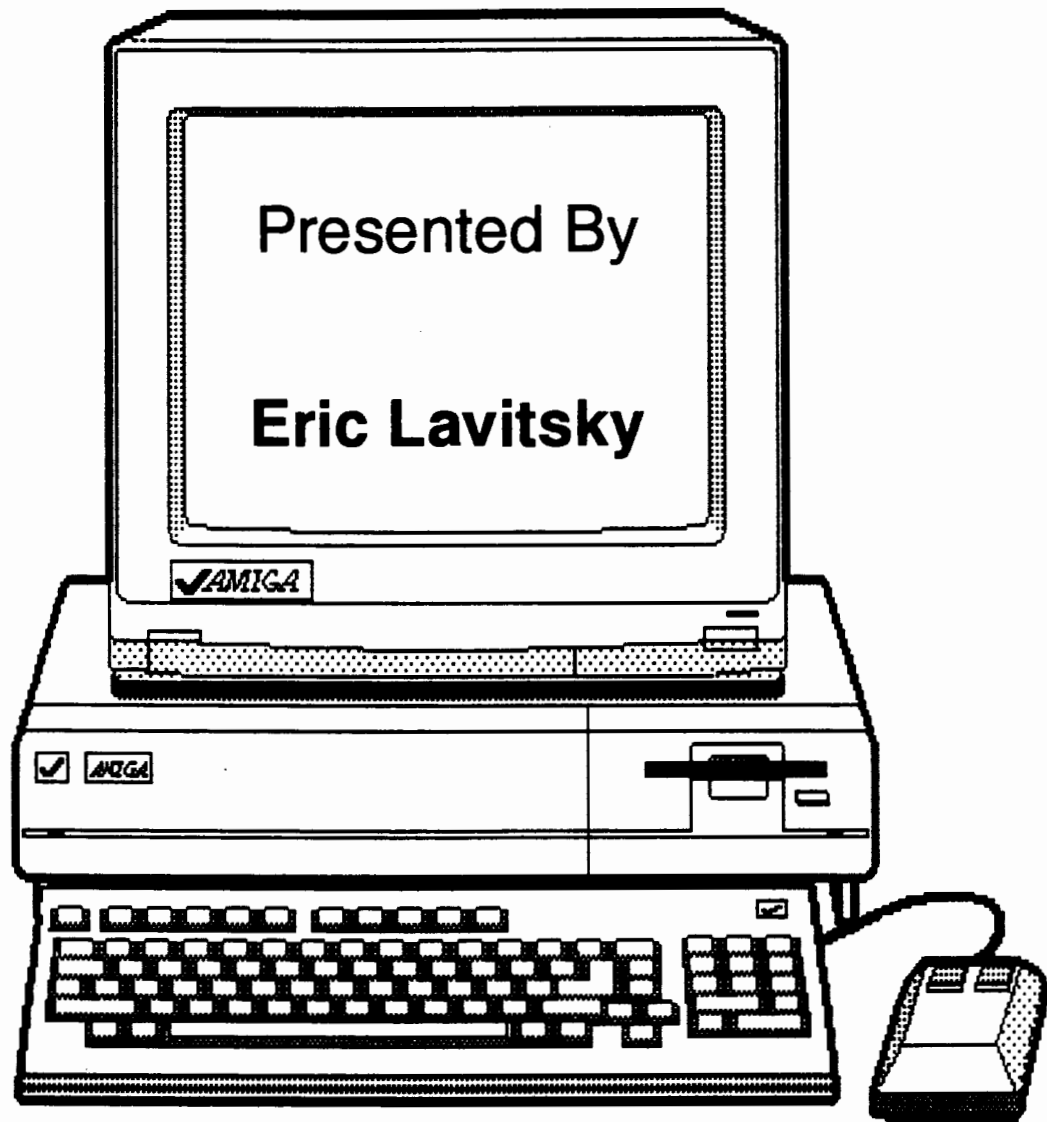
—

—

—



Introduction To Programming The Amiga® Part I



**Copyright © 1988-1993 by Eric Lavitsky
All Rights Reserved Worldwide**

This course is the exclusive property of the author,
Eric Lavitsky, 174 Hyde Park Rd., Somerset, NJ 08873 (908) 560-0106
and may not be reproduced in any way without express written permission.

This course is dedicated to all the wonderful friends and acquaintances I have made over the
past few years working with the Amiga and to everyone who made the Amiga a reality.

Amiga is a registered trademark of Commodore-Amiga Inc.

This document was created entirely on an Amiga A2000/A2500 using PageStream 2.21 from
Soft-Logik and printed on a PostScript Laser Printer.

What Are The Goals Of This Course?

- To provide an understanding of fundamental concepts of Amiga architecture and programming.
- To provide the basic tools to begin writing Amiga applications.
- To provide insight into elements of good (and bad) Amiga programming style.

Prerequisites:

This course assumes a knowledge of 'C' or a similar structured programming language and an understanding of basic concepts in computer architecture. A familiarity with assembly language and other multi-tasking/multi-processor environments is not required, but will be helpful.

Disclaimer:

Many of the examples in this course are presented as fragments or pseudo-code. They were derived from various C Compilers and Assemblers, and tested wherever possible. The author has made every attempt to verify their accuracy, but assumes no liability for their content.

Course Outline

Overview of Amiga Architecture

- Hardware

 - CPU, Custom Chips, RAM/ROM, Expansion

- ROM Kernel

 - Libraries, Devices, Hierarchy

Starting Out With EXEC

- Library Access

 - Library Base Address and Offsets

 - Using The Autodocs

 - Using OpenLibrary/CloseLibrary

 - System Library Summary

- Lists

 - Data Structures and Routines

 - Using Lists

 - Node Types

Advanced EXEC Concepts

- Memory Management

 - Routines

 - Using AllocMem

 - System Memory Organization

 - Memory Pools

- Tasks

 - Data Structures and Routines

 - Launching a Task

 - Controlling Task Scheduling

 - Cache Control

- Signals and Signal Masks

- Events and Wait()

- Message Ports/InterProcess Communication

- EXEC I/O

 - IORequests

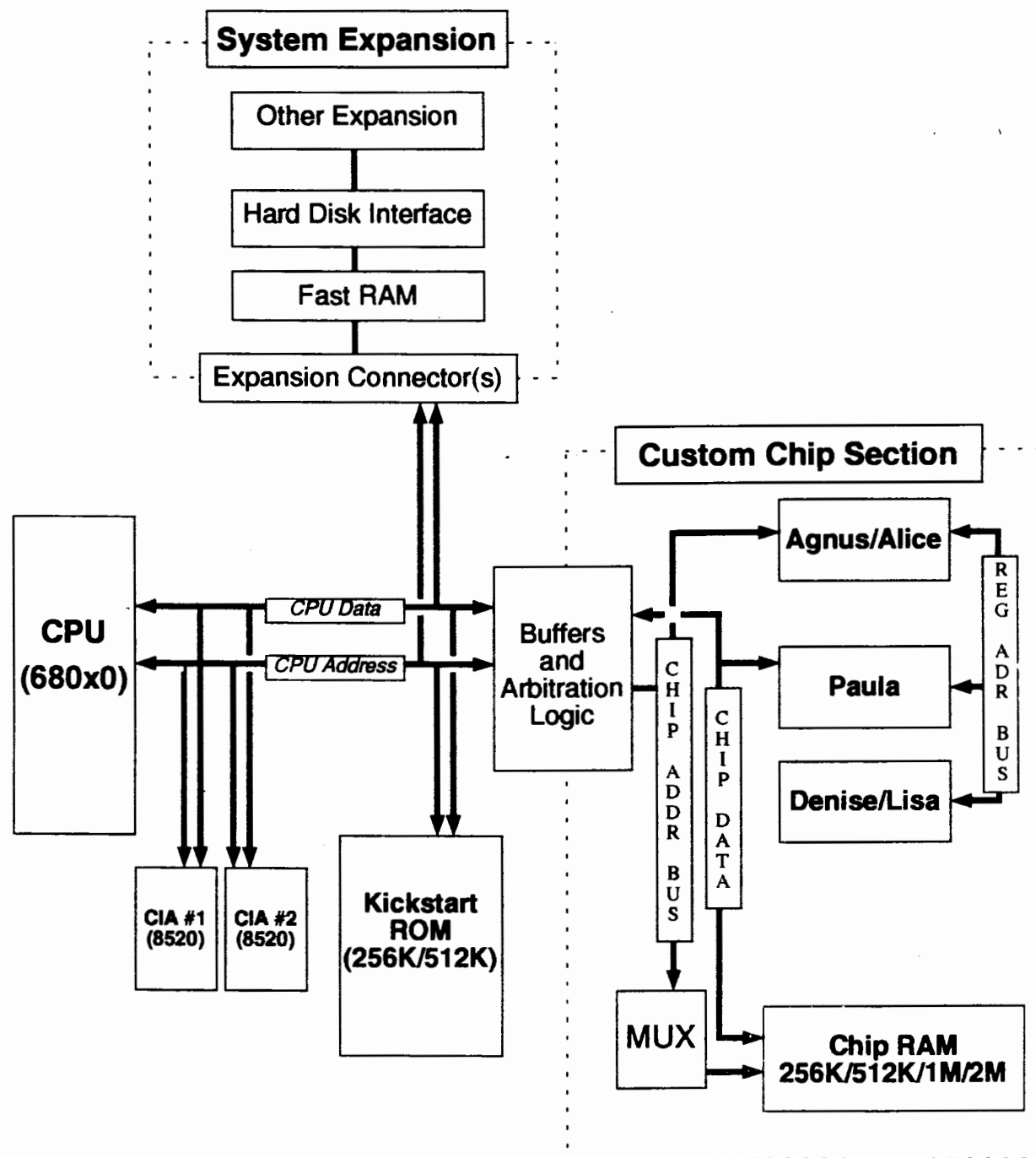
 - Device Example - Using The timer.device

- Guru Meditations and System Errors

Tips and Caveats

References

Amiga Hardware Block Diagram



Hardware In Detail

Agnus/Alice: Copper/CoProcessor

The Copper has 3 instructions:

Move: Move a value into a custom chip register.

Wait: Wait for the video beam to reach a specific position (x,y)

Skip: Skip the next instruction if the video beam has already reached a specific position (x,y).

Denise/Lisa: Color DACs (Digital To Analog Converters)

The registers on Denise determine the color of the display and drive the display itself. Denise is also responsible for control of the hardware sprites.

Paula: Audio Channels and I/O

Paula controls the stereo audio channels and plays a major role in floppy disk control as well as joystick/mouse, serial and parallel port control which it shares with the 8520 CIAs.

Chip Memory/Chip Bus

All the custom chips and a limited addressable range of memory live on this data bus. This bus is also shared with the 680x0 processor.

Fast Memory/Expansion Bus

All add-on memory (above and beyond chip-memory) and devices (e.g. disk controllers, frame buffers) are placed on this bus. The 680x0 accesses this bus without contention (i.e. it runs at full speed at all times). Devices on the Expansion bus may DMA to each other without interfering with devices on the Chip Bus. Expansion devices are mapped above the 8 Meg RAM expansion space. Any addressable memory (e.g. buffer memory) on these devices is configured in 64K chunks.

ROM Kernel

This 256K (1.3) or 512K (2.0) of ROM (Kickstart Writable Control Store on A1000's) contains the instructions that make up the Amiga system software. Without these routines the Amiga would be a useless machine

Amiga ROM Kernel Block Diagram

Libraries

Demand Loadable, Shareable Code, Position-Independent, ROM or Disk

AmigaDOS

(dos.library)

Hierarchical File System
EXEC Based Processes
Multi-Buffer/Disk Operations
Command Line Interface

Intuition

(intuition.library)

User Interface.
Mouse.
Windows.
Screens.
Gadgets.
Menus.

Layers

(layers.library)

Clipping
Regions.
Damage Lists.
LayerOps.

Graphics

(graphics.library)

Display Database.
Manage Copper.
Draw Lines.
Fill Areas.
Blitter.
Text.
GELS.
(Graphics Elements)
BOBs
Sprites
Animation Objects

Devices

Audio	Printer
Clipboard	Ramdrive
Console	SCSI
Gameport	Serial
Input	Timer
Keyboard	Trackdisk
Parallel	

EXEC

(exec.library)
List Processing
Interrupts/Exceptions
Libraries/Devices

Sole Fixed Base Address
Message Based Multitasking
Memory Management
I/O

Hardware

Just What Is A Library Anyway?

A Library is defined as a collection of related routines.

A Library can be either ROM or Disk Based (e.g. graphics.library is in ROM, info.library is on Disk).

A routine in a Library is accessed by jumping to a defined offset from the base address of the Library.

A Library is generally re-entrant - it's code can be shared and executed by more than one task at a time.

A Library is position independent - it can be loaded at an arbitrary location in memory. Individual Library routines may be scatter loaded in memory.

A Library is opened using the EXEC routine OpenLibrary and closed using CloseLibrary.

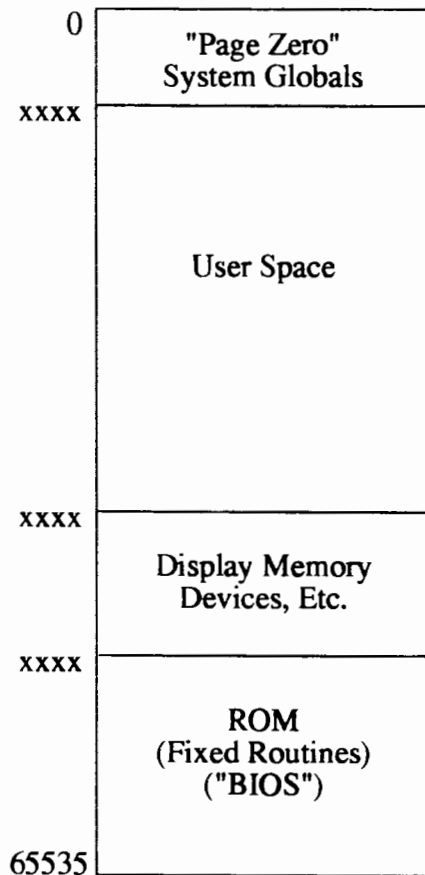
OpenLibrary increments the "usage" count of the Library and CloseLibrary decrements this counter.

When the "usage" or "open" count of a Library reaches 0, EXEC is permitted to flush the Library from memory if the space occupied by the Library is needed by the system and the Library is RAM resident (e.g. loaded from Disk).

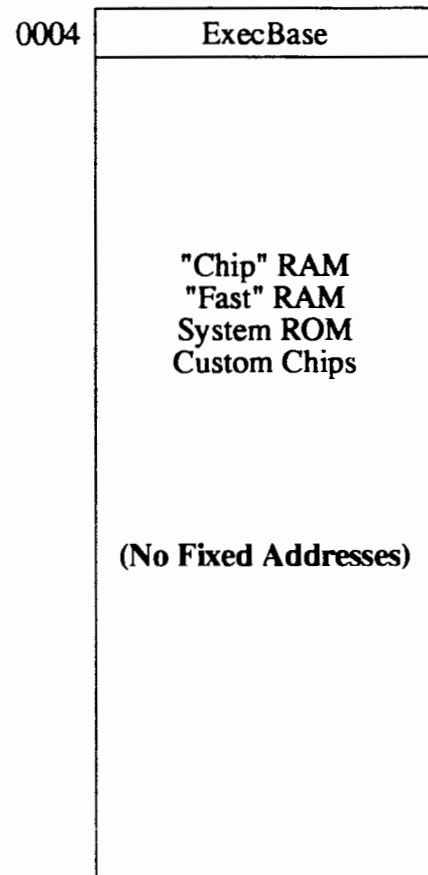
Multiple versions of a Library can be distinguished by a version number (1.2 = V33, 1.3 = V34, 2.0 = V36/V37, 3.0 = V39). Applications should open the lowest version that meets their needs.

Comparison Of Typical PC Memory Map vs Amiga Memory Layout

Typical MicroComputer Memory Map
(Apple II/Commodore 64/IBM PC)



Commodore-Amiga Memory Map



Gaining Access To The Amiga ROMs

```

AbsExecBase      EQU 4      ; normally XREF this
LV0OpenLibrary   EQU -552   ; normally XREF this

IntuitionBase:    dc.l      $0000
ILibName:         dc.b      'intuition.library',0

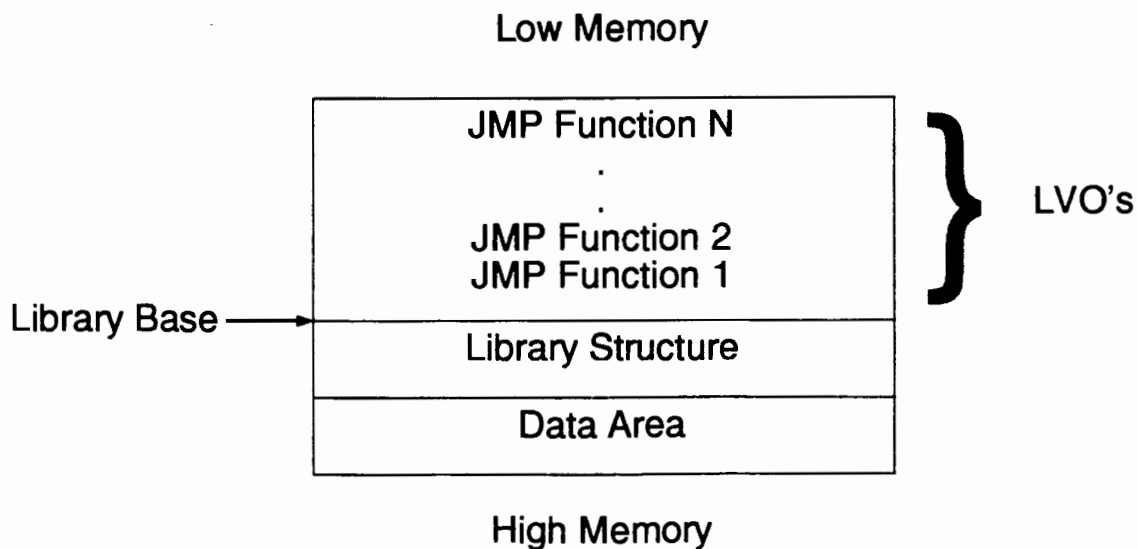
moveq.l          #33,d0      ; want V33 or greater
lea              ILibName,a1  ; get library name
movea.l          AbsExecBase,a6 ; library base ptr
jsr              LV0OpenLibrary(a6) ; call OpenLibrary
move.l           d0,_IntuitionBase ; store the result
beq.s            my_error_abort ; check result
.
.
.

```

This moves the base address (memory location) of ExecBase into register a6 and then calls the EXEC routine OpenLibrary, which is an offset into the exec.library.

Higher level languages do this dirty work for you.

LVO == Library Vector Offset



Using The Autodocs

exec.library/OpenLibrary

NAME

OpenLibrary -- gain access to a library (optional version)

SYNOPSIS

```
library = OpenLibrary(libName, version)
D0                      A1          D0
```

FUNCTION

This function returns a pointer to a library that was previously installed into the system. If the requested library exists, and if the library version is greater than or equal to the requested version, then the open will succeed.

INPUTS

libName - the name of the library to open

version - the version of the library required.

RESULTS

library - a library pointer for a successful open, else zero

SEE ALSO

CloseLibrary

The Autodocs can be an invaluable reference for the latest up to date calling conventions for Amiga routines. Full information on calling conventions for C as well as Assembly and cross reference to related routines is included for every entry.

New functions generally have a version number in parentheses at the end of the name line, e.g: (V36).

Using OpenLibrary()/CloseLibrary()

```
#include <exec/exec.h>          /* Include all the Exec Headers */
#include <intuition/intuition.h> /* Include all the Intuition Structures */
#include <dos/dos.h>

extern struct Library      *OpenLibrary();
struct Library      *IntuitionBase = NULL;

main()
{
    if ((IntuitionBase = OpenLibrary("intuition.library", 33L)) == NULL)
    {
        exit(RETURN_FAIL);
    }

    /* We can now call any Intuition routine */

    if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```

The include files are provided by Commodore-Amiga and are included with all commercial compilers. Their organization follows the logical hierarchy of the hardware and ROM Kernel. Be sure your include files are current with the target version of the operating system you are running.

OpenLibrary() returns the base address of intuition.library if it succeeds. When we exit, we check to see if IntuitionBase does indeed point to a valid library base and we CloseLibrary Intuition if it does.

When we call an Intuition routine, the compiler is smart enough to load the base address of the Intuition Library as reflected in our copy of "IntuitionBase" and call the proper offset into the Library for the routine in question.

Libraries in Detail

When opening a library, you must use the following naming conventions for the library base pointer:

<u>BaseName</u>	<u>Library</u>	<u>Include File</u>
ROM Libraries as of 1.3:		
DiskfontBase	diskfont.library	libraries/diskfont.h
DOSBase	dos.library	libraries/dos.h
ExecBase	exec.library	exec/execbase.h
ExpansionBase	expansion.library	libraries/expansion.h
GfxBase	graphics.library	graphics/gfxbase.h
IntuitionBase	intuition.library	intuition/intuition.h
LayersBase	layers.library	graphics/layers.h, clip.h
MathBase	mathffp.library	libraries/mathffp.h
MathTransBase	mathtrans.library	libraries/mathffp.h
RomBootBase	romboot.library	libraries/romboot_base.h

ROM Libraries introduced in 2.0:

GadToolsBase	gadtools.library	libraries/gadtools.h
KeymapBase	keymap.library	devices/keymap.h
UtilityBase	utility.library	utility/#?.h
WorkbenchBase	workbench.library	workbench/#?.h

Disk Based Libraries as of 1.3:

IconBase	icon.library	workbench/icon.h
MathIeeeDoubBasBase	info.library	not documented
MathIeeeDoubTransBase	mathieeedoubbas.library	libraries/mathlibrary.h
TranslatorBase	mathieeedoubtrans.library	libraries/mathieeedp.h
	translator.library	libraries/mathlibrary.h
	version.library	libraries/mathieeedp.h
		libraries/translator.h
		not documented

Disk Based Libraries Introduced in 2.0:

AslBase	asl.library	libraries/asl.h and aslbase.h
CxBase	commodities.library	libraries/commodities.h
IFFParseBase	iffparse.library	libraries/iffparse.h
RexxSysBase	rexsyslib.library	rex/rxslib.h

Disk Based Libraries Introduced in 2.1:

LocaleBase	locale.library	libraries/locale.h
------------	----------------	--------------------

Disk Based Libraries Introduced in 3.0:

AmigaGuideBase	68040.library	not documented
BulletBase	amigaguide.library	libraries/amigaguide.h
DataTypesBase	bullet.library	diskfont/glyph.h
	datatypes.library	datatypes/datatypes.h

The general rule of thumb is to close libraries in the reverse order from which they were opened.

EXEC Lists

Lists are a fundamental data structure in Computer Science.

Lists are the core data structure of EXEC.

An understanding of Lists is necessary to write programs that deal with all other aspects of EXEC.

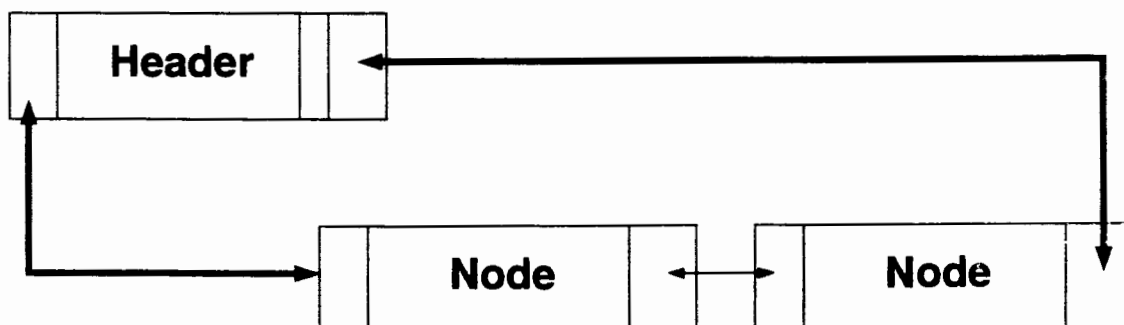
EXEC Lists are extremely efficient.

Your own applications can easily use and benefit from EXEC Lists.

Some ROM Kernel subsystems that directly or indirectly use Lists (Nodes):

- Memory Management
- Tasks
- Interrupts
- Messages/MessagePorts
- Device I/O Requests

Lists can be sorted by name or priority.



EXEC Lists In Detail

List Header

```
struct List {
    struct Node *lh_Head;
    struct Node *lh_Tail; /* always 0 */
    struct Node *lh_TailPred;
    UBYTE lh_Type;
    UBYTE l_pad;
}; /* word aligned */
```

List Nodes

```
struct Node {
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE ln_Type;
    BYTE ln_Pri;
    /* Data Here */
}; /* Note: word aligned */
```

```
struct Node {
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE ln_Type;
    BYTE ln_Pri;
    /* Data Here */
}; /* Note: word aligned */
```

```
struct Node {
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE ln_Type;
    BYTE ln_Pri;
    /* Data Here */
}; /* Note: word aligned */
```

NewList()	Pass this a pointer to a List structure to initialize a List
Insert()	Inserts a node into a list after a specified node already in the list
Remove()	Removes a specified node
AddHead()	Insert a node at the head of a list; more efficient than Insert() for Head
RemHead()	Remove node from the head of a list
AddTail()	Insert a node at the tail of a list; more efficient than Insert() for Tail
RemTail()	Remove node from the tail of a list
Enqueue()	Insert a node into a list in priority order
FindName()	Search a given list for a named node

Note that the Head/Tail is combined. It removes the need for a special case test for the empty list.

You should also think about using MinLists/MinNodes - all these routines except Enqueue() will work with them.

Using Lists

```
#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>

extern void      *AllocMem();
extern struct Node *RemHead(), *FindName();

struct MinList   *MyList = NULL;

struct MyNode
{
    struct Node   INode;      /* This is a Node */
    long          foo;        /* Plus some useful data */
} *IN = NULL;

/* Free the memory used by a MyNode passed in 'IN' */
FreeNode(IN)
register struct MyNode *IN;
{
    if (IN)
    {
        FreeMem(IN, (long) sizeof(struct MyNode));
    }
}

/* Free all MyNodes in a given MinList passed in 'list' */
FreeAll(list)
register struct MinList *list;
{
    register struct MyNode *IN;

    while (IN = (struct MyNode *) RemHead(list))
    {
        FreeNode(IN);
    }
}

/* Free a given MinList passed in 'list' */
FreeList(list)
register struct MinList *list;
{
    if (list)
    {
        if (list->mlh_Head)
            FreeAll(list);
        FreeMem(list, (long) sizeof(struct MinList));
    }
}
```

Using Lists (contd.)

```
/*
 * Our simple example will allocate memory for a MinList and a
 * custom Node structure. It will then initialize the list using
 * NewList(), stuff some data into the Node structure and add it
 * to the end of the MinList using AddTail(). It then checks to
 * see that it did indeed succeed by doing a FindName() on the
 * Node in the MinList. We finish up by freeing *all* the elements
 * (or Nodes) in the MinList and the MinList itself.
 */
main()
{
    struct MyNode *sNode;

    if (!(MyList = AllocMem((long) sizeof(struct MinList),
                           MEMF_FAST | MEMF_CLEAR)))
    {
        printf("couldn't allocate MinList");
        exit(1);
    }
    NewList(MyList);

    if (!(IN = AllocMem((long) sizeof(struct MyNode),
                       MEMF_FAST | MEMF_CLEAR)))
    {
        printf("couldn't allocate MyNode");
        goto die;
    }

    IN->foo = 5L;
    IN->INode.in_Name = (char *) "our node";
    AddTail(MyList, IN);

    if (sNode = (struct MyNode *) FindName(MyList, "our node"))
    {
        printf("found our node");
    }

die: FreeList(MyList);
}
```

Note that by definition we cannot use Enqueue() with MinLists/Nodes

Nodes

EXEC understands some special Node Types (defined in `exec/nodes.h`):

NT_UNKNOWN	Set for foreign node types
NT_TASK	A Task
NT_INTERRUPT	An Interrupt
NT_DEVICE	A Device driver
NT_MSGPORT	A MsgPort
NT_MESSAGE	A Message
NT_FREEMSG	A Message available for reuse
NT_REPLYMSG	A Reply Message
NT_RESOURCE	A Resource
NT_LIBRARY	An Exec Library
NT_MEMORY	A node in the Memory pool
NT_SOFTINT	A Software Interrupt
NT_FONT	A Font
NT_PROCESS	A DOS Process
NT_SEMAPHORE	A Semaphore
NT_SIGNALSEM	A Signal Semaphore
NT_BOOTNODE	An AutoBoot device
NT_KICKMEM	Kickstart Memory
NT_GRAPHICS	
NT_DEATHMESSAGE	

EXEC Memory Management

Remember, there are two types of memory on the Amiga. They are Chip Memory, and Fast Memory.

`AllocMem()`

This routine is the workhorse of EXEC. This routine is used by both the system and applications to allocate individual regions of memory.

`FreeMem()`

The converse to `AllocMem()`, this routine will return memory that was previously allocated back to the system. Be careful that you free only memory which was previously allocated. Under V1.3 and earlier systems, freeing memory not allocated will crash the system with a "Free Twice" Guru Alert (81000009).

`AllocEntry()`

Will allocate multiple regions of memory defined using `MemLists`. These regions can then be managed using `Allocate()` and `Deallocate()`.

`FreeEntry()`

The converse to `AllocEntry()`.

`Allocate()`

Will allocate memory from a given freelist (`MemList`).

`Deallocate()`

The converse to `Allocate()`.

`AvailMem()`

Given a memory type (`MEMF_CHIP` or `MEMF_FAST`), this call will return the amount of free memory of that type in bytes remaining in the system.

`TypeOfMem()`

This routine, given a valid RAM address will return the type of memory referenced by that address (`MEMF_CHIP` or `MEMF_FAST`).

`AllocAbs()`

This routine will attempt to allocate a range of memory beginning at a user specified base address.

Using AllocMem()

```
#include <proto/exec.h>
#include <exec/memory.h>

APTR mydata = NULL;

main()
{
    mydata = AllocMem(128L, MEMF_CHIP | MEMF_CLEAR);
    if (mydata) FreeMem(mydata, 128L);
}
```

Any combination of the following flags (except MEMF_CHIP | MEMF_FAST) may be used to specify the characteristics of the memory to be allocated:

MEMF_PUBLIC: Memory must not be mapped, swapped, or otherwise made non-addressable. ALL MEMORY THAT IS REFERENCED VIA INTERRUPTS AND/OR BY OTHER TASKS MUST BE EITHER PUBLIC OR LOCKED INTO MEMORY! This includes both code and data (e.g. Message Ports). This flag will become significant if the Amiga hardware moves to a Memory Management Unit and/or the software moves to a virtual memory system(or a new flag may be used).

MEMF_CHIP: Only certain parts of memory are reachable by the special chip sets' DMA circuitry. Anything that will use on-chip DMA must be in memory with this attribute. DMA includes screen memory, things that are blitted, audio blocks, raw disc buffers, etc.

MEMF_FAST: Non-chip (expansion) memory. It is possible for the processor to get locked out of chip memory under certain conditions. FAST memory does not suffer from this problem. Since not all machines are guaranteed to have FAST memory, do not insist on allocating MEMF_FAST.

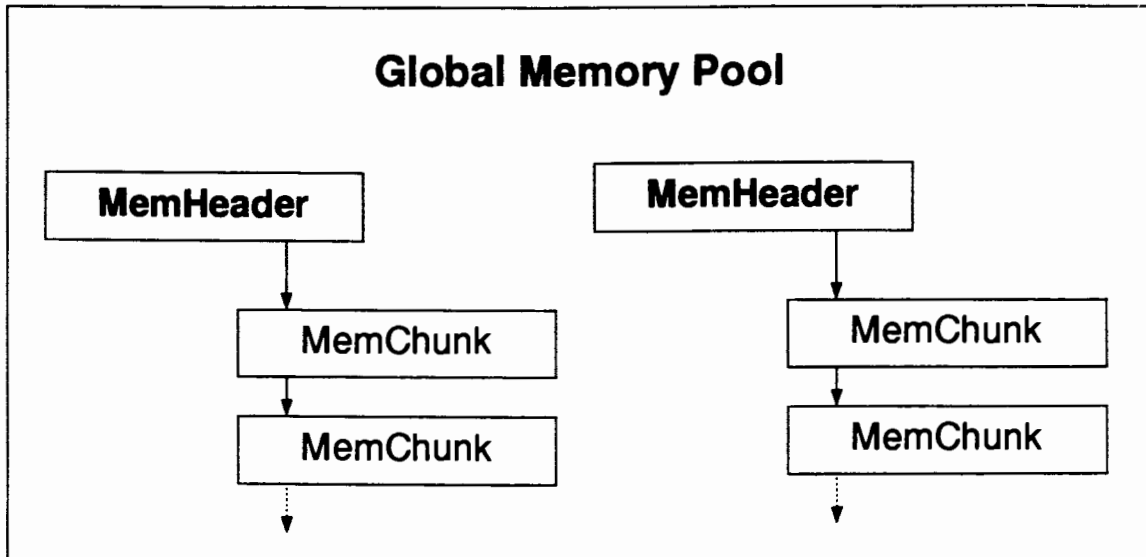
MEMF_CLEAR: The memory will be initialized to all zeros.

AllocMem will always allocate FAST memory if it is available.

New for 2.0:

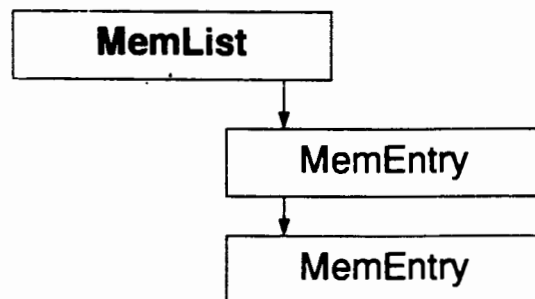
MEMF_LOCAL - Does not go away at reset.
MEMF_24BITDMA - DMA'able in 24-bit address space.
MEMF_REVERSE - Allocate from the top down.

EXEC Memory Organization



The **MemHeader** specifies characteristics (Chip/Fast, etc.), bounds and total free bytes of all the **MemChunks** in the List.

A **MemChunk** is simply a region of memory (# of bytes).



You can allocate multiple memory blocks using `AllocEntry()`, or use the new 3.0 memory pool functions.

WARNING: Read the `AllocEntry()` AutoDoc. The function returns a special non-zero value on failure!

EXEC Memory Pools

Under 3.0, you can use memory pools for more efficient memory allocation and to help reduce fragmentation in the system. While this can be achieved with `AllocEntry()` and its related routines, the memory pool functions hide the underlying data structures from the application. This removes much of the housekeeping requirements from the individual application. It also allows improvements in the underlying memory allocation and management technology without having to re-write individual applications.

Memory Pools are large sections of memory. They are comprised of units called "Puddles". The only thing an application knows about a pool is its address. When a pool is created, you specify the size of the puddles and a threshold value, not the size of the pool. Pools are dynamic - they have no fixed size. As you require memory, the pool manager takes the memory from the puddles. The manager expands and shrinks the pool by adding and deleting puddles as necessary.

```
void *CreatePool(memFlags, puddleSize, threshSize)
    ULONG memFlags, puddleSize, threshSize;

void DeletePool(poolHeader)
    void *poolHeader;

void *AllocPooled(poolHeader, memSize)
    void *poolHeader;
    ULONG memSize;

void FreePooled(poolHeader, memory, memSize)
    void *poolHeader, *memory;
    ULONG memSize;
```

The threshold value is the size of the largest allocation to use within a puddle. An allocation request larger than this value will cause the pool manager to add a new puddle. Ideally, the threshold value should be the size of the largest allocation your application will make.

Threshold sizes cannot exceed puddle sizes and are recommended to be one half the puddle size so that two of an applications largest allocations can fit in a single puddle.

Tasks

A Task is a thread of execution.

Individual Tasks each have their own processor state or context.

Tasks are heavily based on EXEC Lists; each Task begins with a Node.

EXEC completely manages the scheduling of Tasks on the following basis:

The highest priority Task is run and continues executing until either:

- 1) a higher priority Task becomes active.
- 2) the running Task exceeds it's Quantum (~50ms).
- 3) the running Task needs to wait for some external event to occur before it can continue.

The Task that is running is said to be "Running" or on the Run Queue (there is currently only one Task on the Run Queue at any given time).

A Task that is in line to be executed is said to be "Ready" or on the Ready Queue.

A Task that is waiting for an external event is said to be "Waiting" or on the Wait Queue.

When a Task that is Waiting receives one of it's external events, or Signals, it is placed on the Ready Queue.

The Ready Queue is kept in priority-sorted order.

This method of scheduling is known as "Round-Robin Pre-Emptive Task Scheduling"

Prior to 2.0, Tasks could not open disk based libraries or make calls to DOS. Under 2.0 and beyond, a Task may open disk based libraries.

Tasks In Detail

`AddTask()` - Add a Task to the system given an initial and final code address. The Task must have space for it's local stack allocated (200 bytes min recommended for EXEC only, 4K min recommended for dealing with DOS - but your Task will be part of a Process structure).

`RemTask()` - Remove a Task with a given name. `RemTask(NULL)` will remove one's self. Be sure you've freed all allocated resources first!

`FindTask()` - Find a Task with a given name. `FindTask(NULL)` will find the current Task.

`AllocSignal()` - Allocate a signal bit from the Tasks current pool (32 signals per Task or one longword, 16 usable by you).

`FreeSignal()` - Free a previously allocated signal so it may be reused.

`AllocTrap()` - Allocate a processor trap vector.

`FreeTrap()` - Free a trap vector for re-use.

`Forbid()` - Prohibits all Task rescheduling.

`Permit()` - Re-enables Task rescheduling.

`Disable()` - Prohibit all Interrupt processing and Task rescheduling.

`Enable()` - re-enable Interrupt processing and Task rescheduling.

`Wait()` - Relinquish the CPU and wait for one or more Signals.

`StackSwap()` - Replace a Tasks stack.

`CreateTask()` - initialize and launch a new Task (a compiler library routine).

`DeleteTask()` - remove a running Task (a compiler library routine which simply calls `RemTask()`).

The Task Structure

```
struct Task {
    struct Node tc_Node;      /* To link to the next task */
    UBYTE tc_Flags;          /* Task Flags */
    UBYTE tc_State;          /* Task state */
    BYTE tc_IDNestCnt;        /* Interrupt disabled nest count */
    BYTE tc_TDNestCnt;        /* Task disabled nest count (Forbid()) */
    ULONG tc_SigAlloc;        /* Signals allocated */
    ULONG tc_SigWait;         /* Signals we're waiting for */
    ULONG tc_SigRecvd;        /* Signals we've received */
    ULONG tc_SigExcept;       /* Signals we will take exceptions for */
    UWORD tc_TrapAlloc;       /* Traps allocated */
    UWORD tc_TrapAble;        /* Traps enabled */
    APTR tc_ExceptData;       /* points to exception data */
    APTR tc_ExceptCode;       /* points to exception code */
    APTR tc_TrapData;         /* points to trap data */
    APTR tc_TrapCode;         /* points to trap code */
    APTR tc_SPReg;            /* Stack pointer */
    APTR tc_SPLower;          /* Stack lower bound */
    APTR tc_SPUpper;          /* Stack upper bound */
    VOID (*tc_Switch)();      /* See below */
    VOID (*tc_Launch)();      /* See Below */
    struct List tc_MemEntry;   /* allocated memory */
    APTR tc_UserData;         /* per-task user data */
};
```

The `tc_Switch` and `tc_Launch` vectors point to the code that is executed when a task is removed from the run queue (pre-empted) and when a task is put on the run queue (started).

Usually these routines save all the processor registers on your stack during `Switch` and restore them upon `Launch`. It is possible to wedge into these routines to increase their functionality. For instance, if you have a peripheral 68881 under 1.3 or greater, a vendor may supply the necessary routines to save and restore the 68881 registers to be wedged into the `Switch` and `Launch` vectors.

Launching A Task

```
#include <exec/exec.h>
#include <proto/exec.h>

struct Task    *theTask = NULL;
char           *myTaskName = "myTask";
ULONG          myvar;

/* The SubTask waits on a signal from the main program */
void myTask()
{
#ifdef AZTEC_C
    geta4();
#endif

    myvar = 100;
    (void) Wait(0L);    /* Wait forever */
}

void main(int argc, char **argv)
{
    myvar = 0;
    theTask = CreateTask(myTaskName, 0L, myTask, 4000L);

    /* Must synchronize with SubTask 1st This doesn't guarantee it! */
    Delay(5L);
    printf("myvar is: %ld\\n", myvar);

    /* We know it's safe to remove theTask */
    if (theTask) DeleteTask(theTask);
}
```

Note that CreateTask() does the dirty work necessary for calling AddTask(), and DeleteTask() calls RemTask(). CreateTask() also handles all the necessary memory and stack allocation.

The subtask shares the same data space with the parent program.

The subtask may be safely removed if it is not going to be awakened and placed on the ready/run queue (it is waiting for a signal/condition which will not occur).

Enable/Disable - Forbid/Permit

These routines should be used with extreme care. Incorrect use or overuse of these routines can seriously degrade the performance of the Amiga multitasking environment. When should you use these functions?

`Forbid()` is sometimes used when accessing system data structures which are dynamic in nature and which other tasks may access/modify. It will ensure that your task can examine these structures in a consistent fashion. Another mechanism called Semaphores may also be used for this type of activity, and is often preferred. See the `Intuition` routine `LockIBase()` for a case where `Forbid()`'ing is not kosher.

Be aware that calling `Wait()` or calling any routine which may `Wait()` will effectively relinquish the processor to other tasks. When the Task is re-scheduled, you will re-enter the `Forbid()` state (Note that any file I/O and `printf()` cause a `Wait()`).

`Disable()` is similar to `Forbid()`, but also prevents all interrupts from occurring (such as an interrupt from a Disk controller). You might `Disable()` if the data structure you are accessing is shared by Interrupt code. Since so much of the normal activity of the system (e.g. updating the display, controlling the mouse) relies on near real-time interrupts, you should never `Disable()` for more than an instant. Also, do not attempt to call routines that require a multitasking environment while `Disable()`'ed. You will hang the system (e.g. don't call `printf()` or attempt to do other I/O).

Both `Forbid()` and `Disable()` may be nested.

It is never necessary to both `Disable()` and `Forbid()`. Since `Disable()` prevents interrupts, it also prevents task scheduling.

Cache Control Functions

Life was simple back in 1985; a 68000 was a powerful processor. Things have changed a lot since then, and we now have more powerful processors like the 68030 and the 68040. These processors have begun to feature hardware caches. These caches raise many issues when programming a machine that relies heavily on DMA and shared memory for performance as the Amiga does.

There are certain cases where a programmer must explicitly control the state of the Data and/or Instruction caches:

- Writing self-modifying code
- Building jump-tables in memory
- Run-time code patches
- Relocating code for use at a different address
- Loading code from disk

All Amiga system calls that modify instructions flush the cache appropriately (e.g. LoadSeg(), MakeLibrary(), SetFunction()). The following functions can be used to explicitly control the state of the processor caches:

CacheClearE() / CacheClearU()	Clear Caches
CacheControl()	Control Cache Settings
CachePreDMA() / CachePostDMA()	Action prior to and after DMA

The functions accept flags specified in include:exec/execbase.h:

```
#define CACRF_EnableI      (1L<<0) /* Enable instruction cache */
#define CACRF_FreezeI     (1L<<1) /* Freeze instruction cache */
#define CACRF_ClearI      (1L<<3) /* Clear instruction cache */
#define CACRF_IBE         (1L<<4) /* Instruction burst enable */
#define CACRF_Enabled     (1L<<8) /* 68030 Enable data cache */
#define CACRF_FreezeD     (1L<<9) /* 68030 Freeze data cache */
#define CACRF_ClearD      (1L<<11) /* 68030 Clear data cache */
#define CACRF_DBE         (1L<<12) /* 68030 Data burst enable */
#define CACRF_WriteAllocate (1L<<13) /* 68030 Write-Allocate mode
                                     (must always be set!) */
#define CACRF_CopyBack (1L<<31) /* Master enable for copyback caches */

#define DMA_Continue      (1L<<1) /* Continuation flag for CachePreDMA */
#define DMA_NoModify      (1L<<2) /* Set if DMA does not update memory */
```

Signals and Signal Masks

Every task is reserved a ULONG field which identifies 32 possible signals which it can receive. These signals represent some event in the EXEC which may be caused by a software or hardware interrupt. Of these 32 signals, only the upper 16 are usable by your application. The rest are reserved for system use.

So, every bit in the ULONG field uniquely identifies every possible signal for a given task. Signals (bits) from this field are reserved (allocated) with the EXEC system call `AllocSignal()` and made available for re-use with `FreeSignal()`.

`AllocSignal(-1L)` will allocate the next available signal bit. If none is available, it will return -1. In the case below, `AllocSignal()` returns the signal number which was allocated, or 17 (range 0..31).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

31 Before `AllocSignal(-1L)` 16

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

31 After `AllocSignal(-1L)` 16

To re-create exactly which bit was set from this number, we must create a Signal Mask. This is done by taking the longword 1 and shifting it left by the signal number:

0 1
"1L"

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
"1L << 17"

We will need to create masks like this in order to create combinations of signal bits to pass to `Wait()`

Events and Wait()

Signals are used to wake up tasks upon the occurrence of some external event. This is done using the EXEC system call `wait()`. This call accepts a ULONG parameter in which all the signal bits for events to wake up the task are set. So, if we want to be woken up by one event whose signal mask is "signal1", we call `wait()` like this:

```
Wait(signal1);
```

If we wish to be woken up by one or more events, we OR their signal masks together and pass them to `Wait()`:

```
Wait(signal1 | signal2 | ... | signaln);
```

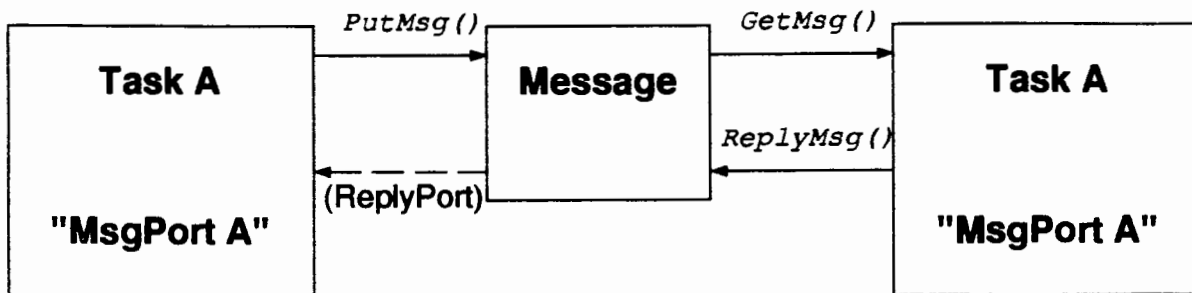
`Wait()` actually returns a ULONG value which represents which signals actually woke the task up. It is the applications' responsibility to determine which signals woke it up and what action to perform.

```
sigsrecvd = Wait(signal1 | signal2 | ... | signaln);  
if (sigsrecvd & signal1) { /* do stuff for signal1 }  
if (sigsrecvd & signal2) { /* do stuff for signal2 }  
if (sigsrecvd & signaln) { /* do stuff for signaln }
```

`wait()` will clear any signals which may have been set in the task block by EXEC. Calling `wait()` a second time will put the task to sleep until the next set of events occur.

When `wait()` is called, the task gets placed on the Wait Queue until EXEC determines that an event has occurred which will wake that task up (based on the signal masks passed to `wait()`). When the task is woken up, the `tc_SigsRecvd` field in the task block gets set by EXEC and EXEC places the task on the Ready Queue.

Message Ports/IPC (InterProcess Communication)



Task A creates it's own uniquely named MsgPort "MsgPortA".

Task B creates it's own uniquely named MsgPort "MsgPortB", then `Wait()`'s on it's Signal Bit.

Task A calls `FindPort("MsgPortB")`.

Task A builds a Message structure, fills it with some data and does a `PutMsg()` to the port returned from `FindPort("MsgPortB")`.

Task A is now free to continue with other processing or to block for a reply from Task B by `Wait()`'ing on the Signal Bits from the Message Reply Port.

Task B, which has been `Wait()`'ing on the Signal Bits for MsgPortB, is woken up by this event.

Task B does a `GetMsg()` to read the message and remove it from the port.

When Task B is finished processing the data in the message, it replies to Task A using `ReplyMsg()`. This tells Task A know that it can re-use the memory from the Message.

The analogy used here is that of an answering machine.

MsgPorts (1.3)

The following function is available in most compiler libraries already. It will allocate a message port with the appropriate signal bits so that it may be used for communication with other processes.

Be sure to use `DeletePort()` before exiting your application!

Be sure to reply to all outstanding messages before calling `DeletePort()` (using `ReplyMsg()`).

```
struct MsgPort *CreatePort(name, pri)
char *name;
BYTE pri;
{
    UBYTEsigBit;
    struct MsgPort *port;

    if ((sigBit = AllocSignal(-1L)) == -1)
        return ((struct MsgPort *) NULL);

    port = AllocMem((long) sizeof(*port), MEMF_CLEAR | MEMF_PUBLIC);
    if (port == 0)
    {
        FreeSignal(sigBit);
        return ((struct MsgPort *) NULL);
    }

    port->mp_Node.ln_Name = name;
    port->mp_Node.ln_Pri = pri;
    port->mp_Node.ln_Type = NT_MSGPORT;

    port->mp_Flags = PA_SIGNAL;
    port->mp_SigBit = sigBit;
    port->mp_SigTask = FindTask(0L);

    if (name != 0)
        AddPort(port);
    else
        NewList(&port->mp_MsgList);

    return (port);
}

DeletePort(port)
struct MsgPort *port;
{
    FreeSignal(port->mp_SigBit);
    RemPort(port);
    FreeMem(port, (long) sizeof(*port));
}
```

MsgPorts (2.0)

Under 2.0 and beyond, EXEC provides new routines to help manage message ports. These are `CreateMsgPort()` and `DeleteMsgPort()`:

```
struct MsgPort *CreateMsgPort(void);  
  
void DeleteMsgPort(struct MsgPort *);
```

You must use `DeleteMsgPort()` for all message ports allocated using `CreateMsgPort()` before exiting your application!

You must reply to all outstanding messages before calling `DeleteMsgPort()` (using `ReplyMsg()`).

If you wish your message port to be added to the public ports list, fill in the `In_Name` and `In_Pri` fields and call `AddPort()` - don't forget to `RemPort()` later! (only ports that will be searched for with `FindPort()` need to be added to the public ports list):

```
msgport = CreateMsgPort();  
  
msgport->In_Name = "MyPort";  
msgport->In_Pri = 0;  
AddPort(msgport);  
.  
.  
.  
RemPort(msgport);  
  
DeleteMsgPort(msgport);
```


Messages

```
struct Message {  
    struct Node mn_Node;  
    struct MsgPort *mn_ReplyPort; /* Port where replymsg will go to */  
    UWORD mn_Length;             /* Length of Message body in Bytes */  
    (Data here)  
}
```

A Message can contain any data you desire to place in one.

Messages live in public memory (MEMF_PUBLIC), they are sent from Task to Task by passing a pointer to their location in memory. This is much faster than copying the whole message body.

Do not re-use a Message until it has been replied to by the recipient.

When you receive a Message, copy it's data to your own local buffers as soon as possible, then reply to it immediately using `ReplyMsg()`.

How To Receive and Act On Messages

Many Messages may queue up at a port at one time, but you will receive only one signal for all of them. You should process all available messages and then Wait for more to come in.

```
for (;;)    /* Loop Forever or until break; */
{
    if ((msg = GetMsg(Port)) == NULL)
    {
        (void) Wait(SignalMask);
        continue; /* Try GetMsg() again */
    }
    process_status = processMsg(msg);
    ReplyMsg(msg);
    if(process_status == QUIT)
    {
        break;
    }
}
```

It is possible to extend this loop to service events from more than one message port:

```
signal = Wait(signal1 | signal2 | ... signaln);
if (signal & signal1) { /* do stuff for signal1 */ }
if (signal & signal2) { /* do stuff for signal2 */ }
...
```

"signal" is what is stored in the tc_SigRcvd field of the Task block. It is a mask of all the signals received since Wait was called. "signal1 | signal2 .. etc." is the mask of signals which EXEC will wake us up for (tc_SigWait).

Before exiting your application, be sure you have replied to all messages waiting on your port:

```
while (msg = GetMsg(Port))
{
    ReplyMsg(msg);
}
```

Launching A Task Revisited

```
#include <exec/exec.h>
#include <proto/exec.h>

struct Task    *theTask = NULL;
char           *myTaskName = "myTask";
ULONG         myvar;

/* The SubTask waits on a signal from the main program */
void myTask()
{
    struct MsgPort *MainTaskPort = NULL;
    struct Message MyMsg;

#ifdef AZTEC_C
    geta4();
#endif

    if (MainTaskPort = FindPort("Rendezvous") == NULL)
        exit(RETURN_FAIL);

    myvar = 100;

    PutMsg(MainTaskPort, &MyMsg); /* Could have just used Signal */

    (void) Wait(0L); /* Wait to be removed */
}

void main(int argc, char **argv)
{
    struct MsgPort *TaskPort = NULL;
    struct Message *Msg;
    myvar = 0;

    if (TaskPort = CreatePort("Rendezvous", 0L) == NULL)
        exit(RETURN_FAIL);

    theTask = CreateTask(myTaskName, 0L, myTask, 4000L);

    /* Must synchronize with SubTask first! */
    Msg = WaitPort(TaskPort);

    printf("myvar is: %ld\\n", myvar);

    if (TaskPort) DeletePort(TaskPort);

    /* We know it's safe to remove theTask */
    if (theTask) DeleteTask(theTask);
}
```

EXEC I/O

```
struct IORequest {  
    struct Message io_Message; /* used by device to return IOReq */  
    struct Device *io_Device; /* Device Node pointer */  
    struct Unit *io_Unit; /* Unit (driver private) */  
    UWORD io_Command; /* Device command */  
    UBYTE io_Flags;  
    BYTE io_Error; /* Error or Warning # */  
};
```

OpenDevice(devName, Unit, ioRequest, Flags)

Open a named device (device names follow the convention "name.device" and live in the DOS DEVS: directory or in ROM) and initialize the IORequest block.

CloseDevice() - close a given device.

All the following functions take an IORequest block as an argument:

SendIO() - initiate an IO command and return immediately (asynchronous)

DoIO() - perform an IO command and wait for completion (synchronous)

CheckIO() - check to see if an IO Request has completed

WaitIO() - wait for the completion of an IO Request

AbortIO() - attempt to abort an IO Request in progress

BeginIO() - perform an IO command either synchronously or asynchronously

I/O completion tells you that an IORequest block may be used again. So the Reply is implicit upon I/O completion.

IORequest Blocks

```
struct IOStdReq {
    struct Message  io_Message;
    struct Device *io_Device;
    struct Unit    *io_Unit;
    UWORD          io_Command;
    UBYTE          io_Flags;
    BYTE           io_Error;
    ULONG          io_Actual; /* Actual length that occurred (bytes) */
    ULONG          io_Length; /* The length of this requested I/O */
    APTR           io_Data;   /* Pointer to data of size io_Length */
    ULONG          io_Offset; /* Offset for block structured devices */
}
```

Various device drivers require variations on the basic IORequest block. The following compiler library routines are available to allocate and free some of the more commonly used IORequest structures:

```
struct IOStdReq *CreateStdIO(port)
    struct MsgPort *port;

DeleteStdIO(iop)
    struct IOStdReq *iop;

struct IOStdReq *CreateExtIO(port, length)
    struct MsgPort *port;
    ULONG length;

DeleteExtIO(iop)
    struct IOStdReq *iop;
```

Under 2.0, the following EXEC routines are provided for managing IORequests:

```
struct IORequest *CreateIORequest(ioReplyPort, size);
    struct MsgPort *ioReplyPort;
    ULONG size;

void DeleteIORequest(ioReq);
    struct IORequest *ioReq;
```

Always do I/O in the largest chunks possible. This will reduce the amount of overhead placed on the system by the Operating System.

Device Example: Using The timer.device

```
#include <exec/types.h>
#include <exec/memory.h>
#include <proto/exec.h>
#include <devices/timer.h>

#define WAIT_TIME      100000L /* Timer delay in microseconds */

extern ULONG           TimerBits;

struct MsgPort         *Timer_Port = NULL;
struct timerequest      *Time_Req = NULL;
BOOL                    topen = FALSE;

InitTimer()
{
    if (!(Timer_Port = CreatePort("Timer Port", 0L))) return(0);
    if (!(Time_Req = (struct timerequest *)
        CreateExtIO(Timer_Port, (long) sizeof(struct
            timerequest))))
    {
        KillTimer();
        return(0);
    }
    if (OpenDevice(TIMERNAME, UNIT_VBLANK, Time_Req, 0L))
    {
        KillTimer();
        return(0);
    }
    topen = TRUE;
    TimerBits = (1L << Timer_Port->mp_SigBit);
    return(1);
}

QueueTime()
{
    Time_Req->tr_node.io_Command = TR_ADDREQUEST;
    Time_Req->tr_time.tv_secs = 0L;
    Time_Req->tr_time.tv_micro = WAIT_TIME;
    SendIO(Time_Req);
}

KillTimer()
{
    if (topen)
    {
        AbortIO(Time_Req);
        WaitIO(Time_Req);
        CloseDevice(Time_Req);
    }
    if (Time_Req) DeleteExtIO(Time_Req);
    if (Timer_Port) DeletePort(Timer_Port);
}
```

Example: timer.device (cont'd)

```
#include <exec/exec.h>
#include <proto/exec.h>
#include <time.h>

extern struct MsgPort      *Timer_Port;
extern struct timerequest  *Time_Req;

ULONG      TimerBits;
LONG       tloc, chip_free, fast_free;
char       date[24] = NULL;

/* This relies on the standard C library routine ctime() */
GetTime()
{
    tloc = time(0L);
    strcpy(date, ctime(&tloc));
    chip_free = AvailMem(MEMF_CHIP);
    fast_free = AvailMem(MEMF_FAST);
}

main()
{
    BOOL Done = FALSE;
    ULONG signals, times = 0;

    if (!InitTimer()) exit();

    GetTime();          /* Get the initial time */
    QueueTime();        /* Queue up a timer request */

    while (!Done)
    {
        signals = Wait(TimerBits);

        if ((signals & TimerBits) != 0L) {
            WaitIO(Time_Req); /* Wait for IO to complete & remove Msg */
            times++;
            if (times == 10)
            {
                Done = TRUE;
                continue;
            }
            GetTime();      /* Get the current time */
            QueueTime();    /* Queue up a new timer request */
        }
    }
    KillTimer();
}
```

Guru Meditations (1.3)

We all get them sometime, but what are they?

The `Alert()` mechanism is in place to warn the user of potential software or hardware problems

An `Alert()` can be "Recoverable" or "Dead End"

A Guru Meditation Alert is almost always a Dead End Alert

The various Amiga-specific alert codes are described in the include file, `exec/alerts.h`. Low numbered Alerts (particularly 2-11,32-47) are standard 68000 exceptions.

A typical Guru Alert looks like this:

```
80038007.nnnnnnnn
```

The first number represents either the 68000 exception vector or the Kernel subsystem which failed. The second number represents the address at which the error occurred. The first number may actually be a combination of numbers OR'ed together, so this alert means:

```
80000000 = This is a Dead End Alert
00030000 = OpenLibrary error
00008007 = on dos.library
```

(This error would certainly be disastrous, and will probably never happen on your system).

Your own application may put up an Alert using the EXEC function `Alert()`, or the Intuition function `DisplayAlert()`.

`Alert()` takes two parameters:

```
Alert(alertNum, parameters)
```

```
alertNum == exception vector or subsystem
parameters == taskid or PC
```


System Errors (2.0)

Same system call as 1.3 (Alert), one parameter:

```
Alert(alertNum)
```

The format of alertNum remains the same

Now called "System Error"

Yellow indicates recoverable alert

Red indicates non-recoverable alert

Alerts now appear as:

```
Error: 0000 0000 Task: 00000000
```

Where the Error is split into the subsystem #/general error and the specific failure code and Task represents the task ID or PC at the time of failure

For Alerts caused by 680x0 exceptions, all registers are copied to a special place in low memory

Certain Alerts which were non-recoverable in 1.3 are recoverable in 2.0 (e.g. FreeTwice)

Topics For Further Study

Devices - console device, serial device, audio device, printer device.

Libraries - diskfont.library, math libraries

Memory Management - CopyMem(), CopyMemQuick().

MultiTasking - Semaphores

References Used For This Course

1. "Amiga ROM Kernel Reference Manual: Libraries and Devices", Commodore-Amiga, Inc., Addison Wesley
2. "Amiga ROM Kernel Reference Manual: Includes and Autodocs", Commodore-Amiga, Inc., Addison Wesley
3. "Amiga Hardware Reference Manual", Commodore-Amiga, Inc., Addison Wesley
4. Autodocs, Commodore-Amiga Inc.
5. Amiga Developer Conference Notes, 1988-1991, Commodore-Amiga Inc.
6. Manx Aztec C Library Source, Manx Software Systems Inc.
7. Motorola M68000 Microprocessor Reference Manual, Motorola Microsystems Inc.
8. The Fred Fish Public Domain Disk Library, Fred Fish

The ROM Kernel manuals can be purchased at most B. Dalton's bookstores or at your local Amiga dealer (support your local dealers!!!)

The Autodocs and Developer Conference Notes can be purchased directly from Commodore-Amiga

Commodore-Amiga Technical Support
1200 Wilson Dr.
West Chester, PA 19380
1-215-431-9180

Manx Aztec C
Manx Software
1 Industrial Way
Eatontown, NJ 07724
1-908-545-2121

SAS/C
SAS Institute, Inc.
SAS Campus Drive.
Cary, NC 27513
1-919-677-8009

Fish Disks
Fred Fish
1346 W. 10th Pl
Tempe, AZ 85281

Also recommended:

The AmigaDOS Manual
Bantam Books, Commodore-Amiga

The Kickstart Guide To The Amiga
Ariadne Software

Programmers Guide To The Amiga
Rob Peck
Sybex Publishing

Fundamentals of Interactive Computer Graphics
J. D. Foley, A. Van Dam

Graphics in Overlapping Bitmap Layers
Rob Pike
ACM Transactions on Graphics 1983

Your Local User Group - Support It!



V39 and AA Compatibility

by Commodore Engineering and CATS

Introduction

The capabilities of the built-in Amiga graphics hardware did not change significantly between the introduction of the Amiga 1000 in 1985 and the release of the Amiga 3000 in 1990. The ECS chipset and the display enhancer added several new graphics modes and increased the functionality of existing modes -- yet the market demanded more.

This document explains some of the features of the latest generation Amiga graphics hardware known as the AA (double-A) chipset, and ways to ensure application compatibility with these and other V39 features and changes. Additionally, coding methods are outlined which will allow current software to automatically exploit some of the new features.

New Hardware Features

The most important capabilities of the new Amiga graphics hardware are summarized below.

- ☐ **Enhanced Bandwidth.** A 32-bit wide data bus allows doubling of Chip memory bandwidth (up to 2x the normal bandwidth) and supports the input of 32-bit wide bitplane and sprite data. Bandwidth may be doubled again (to 4x) by using Fast Page Mode RAM.
- ☐ **More Bitplanes.** The maximum number of bitplanes has increased to 8 in all resolution modes. This translates to a 256-entry color table for each available mode.
- ☐ **Larger Palette.** Each entry in the color table may now be 25 bits wide (8 bits each for Red, Blue, and Green data plus 1 bit for genlock information). This translates to a palette of 16,777,216 colors.
- ☐ **HAM and HALFBRITE** are available in all display resolutions and depths, allowing for greatly enhanced display modes such as 8-bitplane HIRES HAM.
- ☐ **Enhanced Dual Playfield Support.** Each playfield may now have up to 4 bitplanes. The bank of 16 colors in the 256-color table is independently selectable for each playfield.
- ☐ **Enhanced hardware scrolling support.** The resolution of bitplane scrolling has increased to 35ns.

- ☐ **Enhanced Sprite Support.** Sprite resolution can be set to Lores, Hires, or SuperHires, independent of screen resolution. Attached sprites are now available in all modes. However, some new higher bandwidth modes may only allow one sprite (making an attached sprite impossible). Odd and even sprites may use their own independent 16-color bank from the 256-color table. Old format sprites are still 16 bits wide, but new format sprites may be 32 or 64 bits wide. Sprites may now optionally appear in the border region. The horizontal positioning resolution of sprites has increased to 35ns (equivalent to SuperHires pixel widths.)
- ☐ **Hardware scan-doubling support.** 15 kHz bitplanes and sprites may now be scan-doubled for flicker-free display on 31 kHz monitors, and for enhanced display sharing with 31 kHz bitplanes.
- ☐ **ECS compatibility.** New chips will power-up in an ECS compatibility mode, which will allow many older self-booting programs to be run on new machines.

Ensuring Compatibility

This section covers programming techniques that will help ensure that your application will work as expected when running under V39 and on systems that use AA and future generation graphics hardware.

Support of Release 2 is integral to supporting new graphics chips. One of the main reasons is the display database. Starting with the ECS chipset, not all machines have all display modes available. This will be even more true in the future. The only way to know if a particular mode is available is to check the display database.

For a detailed explanation of how to properly open a screen using the display database information to check for available modes, refer to the *Amiga ROM Kernel Reference Manual: Libraries*, 3rd Edition. Also see the Amiga Mail articles entitled "An Introduction to V36 Screens and Windows" (page IV-3) and "Opening Screens and Windows on Any Amiga" (Page IV-17). Some other pertinent details can also be found in the Paris DevCon notes article entitled "Monitors, Modes, and the Display Database."

Once a screen or other display has been opened, all manipulations should also be handled using system calls. In particular, these manipulations must be handled by the system software:

- ☐ Allocation of graphics resources - use the graphics or intuition library
- ☐ Palette manipulation - use the graphics library
- ☐ Manipulating icons - use the icon library

- ☐ Manipulating mouse pointer sprite - use intuition.library to select or change imagery; input.device to move it
- ☐ Manipulating sprites other than the mouse pointer - use the graphics library
- ☐ Allocating and changing colormaps and colortables - use the graphics library
- ☐ Bitplane allocations - use AllocBitMap() under V39 and higher, and AllocRaster() under earlier releases, but please refer to the next section for important information about limitations of AllocRaster().

Furthermore, these manipulations should be handled by the system software whenever possible:

- ☐ Drawing operations - use the graphics library and Intuition library
- ☐ Blitter operations - if possible, use only use higher level blitter functions from graphics.library such as BltBitMap(), BltRastPort(), etc. If you must wait for a blit to complete, always use the system's WaitBlit() function, never your own. The system knows about and handles problems with the BLITTER_DONE signal that different revisions of the Agnus chip have. If you feel that must use the blitter hardware directly (sacrificing all hope for any RTG compatibility), be sure to properly OwnBlitter, then WaitBlit, then use blitter, then DisownBlitter.

Remember that system structures are subject to change or extension. For new graphics hardware, structures likely to be extended are those associated with ColorMaps, ViewPortExtras, Sprites, and many others. To deal with changing system structures, developers should use system calls to allocate, create, and manipulate these structures. Many new "Get" and "Set" calls have been added to graphics.library to provide an upwards-compatible mechanism for reading and setting structure members which were previously accessed by program code either directly or through macros which accessed them directly. For future compatibility, do not directly access or change any structure or value for which a "Get" and "Set" call is available. And do not declare or AllocMem any structure for which a special allocation call is provided.

Most of the new "Get" and "Set" graphics calls for V39 are listed below. Note that some of these functions are workalike replacements for old gfxmacros which poked or peeked a structure, and others provide tag-based capability for getting and/or setting more than one attribute (Attr) of a structure:

SetOutlinePen(rp, pen) (a0, d0)	SetMaxPen(rp, maxpen) (a0, d0)
GetBitMapAttr(bm, attrnum) (a0, d1)	SetWriteMask(rp, msk) (a0, d0)
SetRPAAttrA(rp, tags) (a0/a1)	GetRPAAttrA(rp, tags) (a0/a1)
GetAPen(rp) (a0)	GetBPen(rp) (a0)
GetDrMd(rp) (a0)	GetOutlinePen(rp) (a0)
SetABPenDrMd(rp, apen, bpen, drawmode) (a1, d0/d1/d2)	

Use the newest functions available. For compatibility reasons, it is sometimes not possible to change the behavior of an existing system function to provide enhanced capabilities. In such cases, new functions may be provided. For upwards compatibility, conditionally use the newest functions that are available. For example, when running under an OS earlier than V39, applications that wish to allocate a BitMap should use AllocRaster() and InitBitMap(). New or updated applications should use the new AllocBitMap() call instead when running under V39 or higher. See the V39 graphics library Autodocs for specifications of new functions such as AllocBitMap().

Hardware Compatibility

The new chips power up in an ECS compatible state that allows a large portion of older, self-booting (i.e., game) software to run even though these older titles often access hardware registers directly. However, directly accessing hardware should be avoided by any software that expects to run after the operating system has been started. Keep in mind also that it is difficult to maintain hardware register-level compatibility even on powerup as new and more enhanced chipsets are developed. Avoid directly programming the hardware whenever possible.

In cases where applications *must* access the hardware:

- ☐ Applications must not write spurious data to, or interpret data from, currently unused bits or addresses.
- ☐ Applications must set undefined bits to zero for writes and ignore them for reads.
- ☐ Mask out all the bits except the ones the application is actually interested in.
- ☐ Do not assume the initial state of any registers.
- ☐ Do not read write-only registers or write to read-only registers.
- ☐ Do not read or write *bytes* to the custom chip registers (they are *word* registers).

Graphics and Intuition Issues

The following notes detail some known coding practices that can cause software to function incorrectly on V39 and AA machines.

- ☐ BitMap plane allocations unsuitable for new modes. This problem surfaces because the new chips have an increased fetch bandwidth. The data for each bitplane scan line must be properly aligned in Chip memory for new display modes to work.

For instance, these allocation methods should all be avoided under V39 and higher:

```
/* WRONG for AA */
for(planenum=0; planenum<DEPTH; planenum++)
    bitmap.Planes[planenum]=(PLANEPTR)AllocMem(RASSIZE(width,height),MEMF_CHIP);

/* Also WRONG */
allplanes=AllocMem(DEPTH*RASSIZE(width,height),MEMF_CHIP);

/* Also WRONG */
for(planenum=0; planenum<DEPTH; planenum++)
    bitmap.Planes[planenum]=(PLANEPTR)AllocRaster(width,height);
```

The correct V39 method for allocating rasters is to use `AllocBitMap()` for graphics-level bitmaps and for `CUSTOMBITMAP` screens, or let `Intuition OpenScreen()` or `OpenScreenTagList()` allocate your bitmap. These methods will allow you to get the greater bandwidth and more efficient displays available under V39 in a manner that will be upward-compatible with future enhancements. The new V39 graphics function `AllocBitMap()` properly handles all allocation and alignment issues. The `Intuition OpenScreenTags()` call uses `AllocBitMap()` under V39.

If for some reason your existing code design prevents you from conditionally calling `AllocBitMap()` or `OpenScreen()` when running under V39 and higher, you may be able to at least get compatibility with the currently available higher-bandwidth modes by using `AllocRaster()` or `AllocMem()` after rounding your width up to a multiple of 64 pixels (absolutely no guarantees here; future chips may require greater alignment; we suggest you conditionally code to use `AllocBitMap()` for future compatibility).

- ☐ Incorrect assumptions about `BitMap->BytesPerRow`. Applications that use system functions to initialize and allocate `BitMaps` and their elements should be aware that the value of `BitMap->BytesPerRow` may be different from the expected value, depending on the bandwidth mode chosen, the custom chip type, method of allocation, and the asked-for width of the allocated planes.

Here's the explanation. First, due to fetch-alignment restrictions in AA, some modes require a higher granularity (`BytesPerRow` must be a multiple of 4 or 8, instead of just 2 under ECS). Second, `BytesPerRow` actually means two different things, which have always been identical, but aren't any more when using interleaved bitmaps. `BytesPerRow` officially means "the number of bytes you have to add to a pointer to a byte of the `BitMap` to get to the same place one row down." It no longer can be depended on to mean "the number of bytes in this row."

To detect software compatibility problems related to BytesPerRow changes, it is extremely important to test on a machine with the AA chipset and Mode Promotion turned on in the IControl Preferences editor, and if the software supports various display sizes, to test with sizes which are not divisible by 64. Two typical symptoms of BytesPerRow problems are 1. skewing of the display, where the pixel data for every line is progressively further right or left, giving a diagonal appearance to the display, and 2. images saved with excess blank space at the right.

Note - CATS intends to provide a developer tool, probably called "BumpBPR", which will force all Screen and AllocBitMap() BitMap widths to a multiple of 64 pixels to emulate the higher scanline alignment of higher bandwidth AA modes. This should allow developers to test for many BitMap->BytesPerRow problems on non-AA and no-promotion machines running V39.

- ☐ **Non-matching BitMap->BytesPerRow** Some applications assume that two BitMaps of the same width, but allocated by different methods, will be swappable or block-copyable in various manners. This is often no longer the case. For example, a BitMap allocated by OpenScreen (which calls AllocBitMap) may have raster line storage widths rounded up to provide higher alignment for higher-bandwidth displays. But the raster lines of BitMap planes allocated with AllocMem() are generally rounded up only to a multiple of 16 with a hardcoded macro (RASSIZE), and for compatibility reasons, AllocRaster() currently still performs only the same rounding to a multiple of 16.
- ☐ **Interleaved BitMaps** For interleaved bitmaps, BytesPerRow is quite a bit larger than the number of bytes in this row. Screen grabbers and screen printers seem to be the primary victims of this change. For compatibility, the Workbench screen is non-interleaved if it opens before IPrefs has run. If opened or reset after IPrefs has run, the Workbench screen will be interleaved, so under V39, the Workbench screen in a normally booted environment is likely to be interleaved. GetBitMapAttr(bm, BMA_WIDTH) can return the true width of a BitMap in pixels. However, this is not the width to be used when saving a BitMap as an ILBM since this width may be rounded up for alignment reasons.
- ☐ **Incorrect assumptions about the internal structure of BitMap plane data.** New types of BitMaps may have a significantly different layout. For example, new V39 interleaved BitMaps are allocated as one contiguous block of Chip memory and are internally different from non-interleaved plane data.

Interleaved BitMaps consist of line 0 of plane 0, followed by line 0 of plane 1, line 0 of plane 2, on through to line 0 of plane n. Then the pattern repeats with the data for the next line. This BitMap layout reduces the color flashing effect which normally accompanies the blitting of individual planes. Interleaved BitMaps can be requested by applications under AA. Note however, that callers are not guaranteed to receive an interleaved BitMap whenever they ask for one. The BMF_INTERLEAVED flag to AllocBitMap() is considered a request, not a requirement. If no sufficiently large continuous block of chip memory is available, it may not be possible to allocate the BitMap interleaved. In that case, AllocBitMap() will attempt to allocate an ordinary BitMap instead. Applications should be written so as not to be sensitive to whether or not a BitMap is interleaved. In those rare cases when it might matter, GetBitMapAttr(BMA_FLAGS) can be used.

- ❑ Incorrect assumptions about the Display Database. Data about a display mode could change between one version of Kickstart and another, between different models of the Amiga and even between similar models. Treat the data in the display database as dynamic! Information that was available under V37 may not necessarily be available under V39. For instance, VGA and A2024 mode information is no longer in ROM and unless they are added by the user, there will be no information about these modes available.

Never cache information about the DEFAULT_MONITOR_ID modes because the default monitor can be changed by the user with the promotion feature.

Use the 2.1 asl.library screen mode requester to present your user with choices which are actually available (Note - 2.1 asl.library works under 2.0 Kickstart, and a free amendment to distribute 2.1 asl.library is available for 2.0 Workbench licensees).

- ❑ Direct handling of ViewPorts. Applications that use low-level graphics calls to create the View directly may have problems in V39. MakeVPort() and MrgCop() can now return an error. In addition, the gap required between ViewPorts may now need to be significantly larger than on pre-AA chips, where it was never more than 3 non-interlaced lines (6 interlaced lines). The V39 graphics function CalcIVG() (Calc Inter Viewport Gap) may be used to determine how many blank lines will be needed above a ViewPort. Note that Intuition currently uses at least 3 lines, or MAX(3,CalcIVG(v,vp)) (substitute 6 for 3 if interlaced). Also note that for old display modes with old depths and 4-bit-per-gun colors, the gap is unchanged from that under the ECS and original chipsets.

- ❑ **Incorrect use of PAL or NTSC flags.** The common method of determining whether a machine is running in PAL or NTSC mode has always been to check for the PAL bit in GfxBase->DisplayFlags:

```
BOOL IsPAL;
```

```
IsPAL=(GfxBase->DisplayFlags & PAL) ? TRUE : FALSE;
```

One classic reason for desiring such a PAL/NTSC determination is to decide what size or type display to open. The other common reason is to determine what clock constant to use in serial or audio period calculations (Amigas built as PAL machines have a slightly different system clock crystal frequency than Amigas built as NTSC machines, and serial/audio period calculations are dependent on this frequency).

Since V37 and V39 are more adaptable, this bit may no longer be counted on to mean the user's preferred display mode, nor to even signify the probable clock crystal in the user's system.

For accurate serial and audio period calculations, you need to determine whether the system has a system clock crystal designed for PAL or NTSC. There is no totally foolproof way of doing this. Under 1.x through 2.x, your best indicator of whether a system has a PAL system clock crystal is the GfxBase->DisplayFlags PAL bit. This bit should tell you the hardware default of the machine, and should only mislead you if the user has moved the PAL/NTSC jumper or has run a PD program to change the bootup mode between PAL and NTSC.

Under V39 (and probably above), new BootMenu options now allow the user to software-select PAL or NTSC as the default graphics environment. This can modify the GfxBase->DisplayFlags PAL bit, but obviously does not change the system clock crystal. Therefore, under V39 and higher, a new DisplayFlags bit called REALLY_PAL has been added to signify the whether the motherboard hardware jumper setting appears to be PAL. Therefore, under V39 and higher, the best bit available for determining NTSC or PAL hardware is the new REALLY_PAL bit.

If your software needs to determine the characteristics of default displays, prior to V37 look at DisplayFlags PAL bit. Under V37 or higher, instead use GetDisplayInfoData() to get information about the modeid LORES_KEY or HIRES_KEY, and then check the DisplayInfo.PropertyFlags for properties such as DIPF_IS_PAL (which will be true for both PAL and DblPAL).

If your software instead needs to determine the characteristics of the user's preferred display mode (as evidenced by the mode of her Workbench or other default public screen), you can GetDisplayInfoData() on the modeID of the default public screen. Be aware that this modeID *might not* be any of the PAL or NTSC modes, but rather VGA or some other mode.

```

#include <graphics/displayinfo.h>

Bool IsPAL;
struct Screen *screen;
ULONG modeID = LORES_KEY;
struct DisplayInfo displayinfo;

/* Open graphics.library, etc. */

/* Above, modeID is initialized to LORES_KEY. You should inquire about LORES_KEY
 * or HIRES_KEY if you are interested in the display characteristics of default
 * displays (for example, what OpenScreen() will provide if you don't use tags to
 * ask for a display mode with an explicit monitor like PAL or NTSC or VGA).
 *
 * Do the following LockPubScreen part if you are instead interested in the display
 * characteristics of the modeID of the user's default public screen (usually
 * Workbench).
 */

if (screen = LockPubScreen(NULL))
{
    modeID = GetVPMODEID(&(screen->ViewPort));
    UnlockPubScreen(NULL, screen);
}

/*
 * Now get information about the modeID
 */
if (GetDisplayInfoData(NULL, (UBYTE *) &displayinfo, sizeof(struct DisplayInfo),
    DTAG_DISP, modeID))
{
    /* True if PAL or DblPAL */
    if (displayinfo.PropertyFlags & DIPF_IS_PAL)
        IsPAL = TRUE;
    else
        IsPAL = FALSE;
}

```

- ❑ **Incorrect assumptions about ColorMaps and ColorTables.** The color system has undergone significant changes so new V39 graphic functions should be used to manage color whenever possible. ColorMaps should always be allocated using `GetColorMap()`; freed using `FreeColorMap()`; colors changed using `LoadRGB4/32()`, `SetRGB4/32()`, or `SetRGB4/32CM()`; and colors queried using `GetRGB4/32()`. Specifically, the value `ColorMap->ColorTable` and the structure it points to should never be poked or read directly.

The color functions with names containing "32" are new V39 functions for getting, setting, and loading color registers. These new functions treat color guns (R, G, B) each as 32-bit values for handling not only the 8-bit color available in AA but also any conceivable future needs. Use of these new 32-bit color functions is required to display the 16 million different color shades available under AA and V39. The old 4-bit color manipulation functions can only provide 4096 different colors. Use these new 32-bit color functions to ensure the future compatibility of your V39 applications.

When using 4 or 8-bit R/G/B values, scale your values to 32 bits. Scale 8 bit R/G/B values to 32 bits by duplicating the 8-bit value in all 4 bytes of the 32-bit value. Scale 4-bit values to 32 bits by duplicating the 4 bit value in each nibble of the 32-bit value.

4-bit red value \$3 becomes \$33333333

8-bit red value \$1F becomes \$1F1F1F1F

8-bit red value \$03 becomes \$03030303

The graphics.library VideoControl() function should be used to get, set, or clear the special features associated with ColorMaps.

- ☐ Poking Copper lists or copinit. The structure and order of Copper lists will change for all modes, so any program that relies on poking the Copper lists will likely break. Certain programs (especially games) make assumptions about copinit and poke it directly. These programs will break with the AA chips and V39 unless the AA machine is running in a non-AA old-chip emulation mode. Note again that copinit has changed, and will continue to change. Do Not Touch.
- ☐ Dangerous bits in the display hardware. Many bits in the custom chip registers that were previously undefined now have a function. Beware of poking the following bits.
 - ☐ In BPLCON0: bits 0, 4, 5, 6, and 7
 - ☐ In BPLCON2: bits 7, 8, 9
 - ☐ In BPLCON3: bits 0,1,6,7, and bits 9 through 15
- ☐ Illegal use of the processor to write to bitplane and sprite data registers. Bitplane and sprite data registers should NOT be written to by the processor. The correct way for data to be fed to the chips is by setting up DMA that points to the data in Chip memory.
- ☐ Illegal re-use of sprites on the same line. Attempting to re-use sprites on the same horizontal line by redefining the sprite data is illegal and unsupported. Sprites may only be re-used vertically, and then only with at least a one line gap between re-uses. See the *Amiga Hardware Reference Manual* for an example of sprite re-use.
- ☐ Incorrect allocation of sprite image data. Because of the increased fetch bandwidth of the new chips, sprite image data must be properly aligned in Chip memory. Under 2.0, the best way to allocate this memory is to call the Exec library AllocMem() function. Use the new AllocSpriteData() call for allocation of new mode sprites under V39 and higher.
- ☐ Using an attached sprite for the Intuition pointer was quasi-supported under 2.0. It no longer works.

- ❑ **Forgotten FreeSprite() call.** Sprites may now have a number of different modes. However, under Intuition, these modes are global to all sprites. If a program gets a sprite in a particular mode, but then does not free it, Intuition (and all Intuition-based programs including Workbench) are forced to use sprites in the mode of the forgotten sprite. In general, it is good practice to not use sprites other than the pointer when coding software that is meant to be Intuition-compatible.
- ❑ **Illegal use of Intuition pointer data.** Intuition's pointer handling has become much more sophisticated. People playing tricky games with the pointer data may get into trouble if they make assumptions about the data format of the Intuition pointer which can change depending on the display mode.
- ❑ **The old SetPointer() and ClearPointer() functions still work,** giving compatible Intuition pointers. Likewise, the pointer imagery in devs:system-configuration will be used until a pointer.prefs file is received. However, due to growing complexity in the pointer subsystem, calling SetPointer() or ClearPointer() from within an input handler or inside Begin/EndRefresh() runs a risk of deadlocking. There are currently some patches in place to help prevent a deadlock, however we warn people to stick to simple rendering functions only when inside LockLayer(), LockLayerInfo(), BeginRefresh(), BeginUpdate(), etc. and not to call Intuition or other high-level system functions inside of LockIBase(). As well, great care should be taken inside an input handler (it's generally best for the handler to signal a high-priority task which actually does the work). Don't be the application that dies under the next release when an inappropriate function that happened to work now deadlocks because its handling has become more sophisticated. See the Autodocs for these functions for more information on what other system calls are OK to use with these functions.
- ❑ **The pointer information returned by GetPrefs() is no longer kept up-to-date,** since the pointer data can exceed the storage space available in struct Preferences. (The ROM default pointer will be returned in all cases). (Like V37, V39 ignores the pointer data in calls to SetPointer() after the first one, for reasons such as this).
- ❑ **OpenScreen failure if too deep.** Since all modes and all depths are no longer available on every systems, OpenScreen will fail if you request a mode/depth combination that is not displayable. This can be an issue both for applications that may have been counting on allocating extra planes this way, or counting on having extra planes that would not be displayed. There is a new SA_ErrorCode value, OSERR_TOODEEP.

- ❑ **Intuition and Graphics are involved in a scheme to maximize compatibility with older sprite-using applications.** The AA chipset supports variable sprite resolution and width, but the setting affects all sprites. If an application requests a specific sprite width (through the old `graphics.library/GetSprite()` call or the new `graphics.library` calls (`GetExtSpriteA()` and `ChangeExtSpriteA()`), and Intuition's sprite is not compatible with that request, then `graphics.library` will blank Intuition's sprite and notify Intuition. Intuition will rebound by generating the most suitable pointer sprite which is compatible.
- ❑ **Pens for Screens.** The handling of defaults for the pen array is a bit involved because of compatibility issues. The only change for old applications should be those who pass an `SA_Pens` array of `{~0}`. They will get the new values for `BARDETAILPEN`, `BARBLOCKPEN`, and `BARTRIMPEN`. This will only affect the color of their screen title bar, and the color of the menus of any window on that screen which requests new-look menus (`WFLG_NEWLOOKMENUS`). Applications that specify no `SA_Pens` array or ones who specify an `SA_Pens` array with at least one explicit pen provided get V37-compatible defaults. Our options were limited because the request for default palette and default pens were too loosely coupled (until `SA_LikeWorkbench`).
- ❑ **Screen.BitMap is becoming obsolete.** The original `Screen` structure has an embedded instance of a `BitMap` structure, which can not grow to support current needs. When Intuition allocates a on-CUSTOMBITMAP screen's `BitMap`, it now uses `AllocBitMap()`, and just copies the old struct `BitMap` portion into the embedded `&screen->BitMap`. Intuition internally now references the true `BitMap` (obtainable as `screen->RastPort.BitMap`) instead of `&screen->BitMap`. We recommend that applications do the same. This is the direction we're headed anyway for RTG, and is needed for double-buffering. There are currently some patches in Intuition to handle people who are poking the `Screen.BitMap` depth or planes. Such poking is strongly discouraged. See the new V39 Intuition function `ChangeScreenBuffer()`.
- ❑ **Console compatibility** Applications which draw graphics in the console portion of the window, but use console scroll commands may not work if they are drawing their text in other than pen 1 or pen 0 when only the bitplane for those colors is scrolled.

Applications that use pen sharing on their own screens should allocate and set pens 3-7 if they plan on using those colors in console. The console device still uses pens 0-7 just as it always has.

Promotion-Related Issues

Under V39 and AA, when Mode Promotion is turned on, display modes which use the default monitor (e.g., display modes that are not explicitly PAL or NTSC) are promoted to scan-doubled or de-interlaced modes such as DbIPAL or DbINTSC to provide a flicker-free display. Mode Promotion, on or off, is controlled system-wide by the IControl Preferences editor and the availability of the DbINTSC or DbIPAL monitor.

Mode promotion can change the behavior of the system in a manner which may be incompatible with certain applications.

- ☐ The overscan limits of DbINTSC (DbIPAL) are a little less than the overscan limits of NTSC (PAL). (For 3.01, the DbINTSC (DbIPAL) limits have been extended and are now comparable to, but still less than, NTSC (PAL)).
- ☐ It may be harder to center a DbINTSC or DbIPAL screen on certain multiscan monitors than it was to center a hardware de-interlaced NTSC or PAL screen. (3.01 provides additional centering flexibility which helps solve this problem).
- ☐ An interlaced screen is promoted to a non-interlaced screen, which has obvious implications on custom copper-lists and copper-list pokers.
- ☐ The higher resolutions/depths of the AA chipset require higher alignment restrictions on bitplanes. Fortunately, most applications either let Intuition allocate their screen's BitMap or else they have a custom BitMap whose width is a multiple of 64 pixels (the highest alignmentoverscan limits of NTSC (PAL). (For 3.01, the DbINTSC (DbIPAL) limits have been extended and are now comparable to, but still less than, NTSC (PAL)).
- ☐ It may be harder to center a DbINTSC or DbIPAL screen on certain multiscan monitors than it was to center a hardware de-interlaced NTSC or PAL screen. (3.01 provides additional centering flexibility which helps solve this problem).
- ☐ An interlaced screen is promoted to a non-interlaced screen, which has obvious implications on custom copper-lists and copper-list pokers.
- ☐ The higher resolutions/depths of the AA chipset require higher alignment restrictions on bitplanes. Fortunately, most applications either let Intuition allocate their screen's BitMap or else they have a custom BitMap whose width is a multiple of 64 pixels (the highest alignment currently required by AA). However, if the custom BitMap is an unusual width, it may not be sufficiently aligned for the hardware. Such a screen can come up skewed when promoted.

- ❑ Coercion of displays in back to match a promoted or explicit DblINTSC or DblPAL front display may result in a lower resolution mode for unsuitably aligned back displays.

“1x” modes require 16-pixel (word boundary) alignment of each scan-line. “2x” modes require 32-pixel (longword boundary) alignment, while “4x” modes require 64-pixel (double-longword boundary) alignment. Here is a short reference:

- ❑ 140 ns pixels (lores in 15 kHz modes, extra-lores in 31 kHz modes)
1-8 planes require 1X
- ❑ 70 ns pixels (hires in 15 kHz modes, lores in 31 kHz modes)
1-4 planes require 1X
5-8 planes require 2X
- ❑ 35 ns pixels (super-hires in 15 kHz modes, hires in 31 kHz modes)
1-2 planes require 1X
2-4 planes require 2X
5-8 planes require 4X

As the graphics.library AllocBitMap() function takes care of allocating suitably-aligned BitMaps for you, you do not need to worry about alignment when using modern system calls.

- ❑ The AA hardware does not allow dual-playfield non-interlaced screens to be scan-doubled, so they will appear half as tall as their non-promoted counterparts.
- ❑ Like earlier chipsets, the AA chipset still supports eight sprites. In much the same way as ECS and original chipsets lose sprites when overscan is increased, many of the new modes have insufficient spare cycles to fetch data for these sprites. A promoted screen may have fewer sprites left than the corresponding 15 kHz mode, meaning that some sprites other than the pointer sprite may vanish.

Note - If the system is aware that additional sprites are needed, it will attempt to drop a display's bandwidth (if possible) to allow additional sprites. To cause this, either GetSprite() your sprites before opening your display, or RemakeDisplay() after doing your GetSprite() calls.

- ❑ There is currently no 31 kHz mode having 1280 pixels per line. That would require 17.5 ns pixel speeds, which is twice what the AA chipset is capable of. Therefore, SuperHires screens are promoted to 640 pixel-per-line screens, which generally can scroll. This is required for systems that have a VGA-only monitor, but otherwise might not be the desired way to display this mode. To prevent promotion of

SuperHires to Hires, an application could explicitly ask for NTSC or PAL SuperHires. However, keep in mind that future chipsets may be capable of de-interlacing 1280-wide displays, and this would be defeated if NTSC or PAL is explicitly requested. Also remember that a user with a VGA-only monitor cannot display these modes. The `V39BestModeIDA()` function may be used to determine the best available mode.

- ❑ Applications that specifically request the NTSC or PAL monitor when opening a display will receive non-promoted PAL or NTSC, suitable for genlocking, etc. For interlace modes, this will not be a flicker-free display. Therefore, productivity and design applications that wish to offer an explicit PAL/NTSC choice should use the display database or 2.1 ASL screen mode requester to allow the user to choose a flicker-free display mode that is available on their system.

Other Compatibility Issues

- ❑ Library Vector Offsets and `SetFunction()`. In general, some new LVOs supersede old ones. While the old ones still work, software that calls `SetFunction()` based on the old LVOs may no longer catch what they were hoping to. An example from V37 was people who called `SetFunction()` on `AutoRequest()`. Most system requests go through `EasyRequest()` now. A V39 example would be `SetWindowPointer()` replacing `SetPointer()`.
- ❑ Test for Memory by Poking Breaks. Apparently, some games test for RAM at \$200000 by writing a pattern there and immediately reading it back. On the A1200 bus, such games will be fooled. Never test for RAM by poking/peeking. Always use Exec's memory allocation functions, or the `Exec TypeOfMem()` function which can tell the caller whether (and what type of) memory exists at any particular address.
- ❑ Changes to support for width-scaling of fonts Under V37, different widths of one YSize of a scalable font could be opened with a tagged `OpenDiskFont()` by first running a patch called `setpatchwtam`. Under V38, this patch program does nothing, and an incorrect already-opened width may be returned. Under V39, the correct requested width should again be returned, and these loaded width-scaled fonts are both hidden from `AvailFonts()` and should not be accidentally provided if application just does `OpenFont()` for the same YSize.
- ❑ Tablet driver writers should test that their tablets now work with console drag selection (known not to work in the 2.04 OS).
- ❑ The V39 `Iffparse.library` is not 1.3 compatible. Use the V37 library instead.

- ❑ **Preferences file format changes** The format of some Preferences files have changed of necessity to support new system capabilities, and these files may continue to change. The documentation which has been provided only shows some chunks which may be encountered in a system Preferences file. There is no guarantee that new chunks will not be added, or that the current chunks will continue to be used. Do not read or write system Preferences files. Instead use other provided system mechanisms for reading and setting such items (for example, use device-specific commands or structures for controlling device preferences, functions such as `QueryOverscan()` and `LockPubScreen()` for determining display characteristics, etc.)
- ❑ **Self-modifying, copied, and directly loaded code** Self-modifying code without proper cache control has been breaking since the 68020 was introduced. The larger caches of the 68030 and 68040 processors make it even more necessary to use the exec cache control functions such as `CacheClearU()` which have been available since V37. The cache control functions make it possible to flush the processor caches after modifying code in place, copying code to a different memory address, or placing code into memory via any mechanism that bypasses `LoadSeg`. Cache-control functions are also provided for DMA controllers which DMA data into memory.
- ❑ **IFF code.** Older IFF sample/example code contains many hardcoded limits and accesses some structures which may no longer be accessed directly. The older code (and even the code listed in the *Amiga ROM Kernel Reference Manual: Devices*, 3rd Edition) also would not load more than 32 color registers. The new IFF code, `NewIFF39.lzh`, attempts to use the latest system functions wherever possible in a conditional backward-compatible manner. The new code provides support for the full AA color palette, arbitrary display modes, the clipboard, overscan and autoscroll screens, and loading/saving of 24-bit ILBMs. See the ILBM Compatibility notes for additional information on ILBM compatibility and support of AA features.
- ❑ **SetMap is gone.** The `SetMap` command (that set the keymap for the whole system) has been replaced by `SetKeyBoard`, that only sets the keymap for the current console. However, if you absolutely require it, the V37 `SetMap` command still works.
- ❑ **Speech is gone.** The `narrator.device` and `translator.library` are not part of V38 and V39. Related handlers and utilities are also gone. Therefore, new machines shipped with V38 or V39 do not have speech.
- ❑ **Pre/Post DMA Cache functions and 68040** For those of you with DMA devices that need to use the `CachePreDMA()` and `CachePostDMA()` calls: Be careful as incorrect use of these calls may look like they work

fine in 68000/68020/68030 systems but may cause strange system behavior and even major system slowdown on 68040 systems.

The correct way to use these calls is:

```
original_length=length;
CachePreDMA(address,&length,0);

/* Optional, multiple ones here for breaking up DMA operations within the single
 * larger DMA block defined in the first call to CachePreDMA()
 */

CachePreDMA(.....,DMA_Continue);

/*
 * Very important to call CachePostDMA() with the exact same values as
 * CachePreDMA()
 */

CachePostDMA(address,&original_length,0)
```

Also, you must make sure they match up. If they do not, the system will not be able to figure out that the DMA that was starting has finished. Internally, the OS needs to track these things for the 68040 and higher processors and may need to track them even more in the future.

- ☐ An upcoming release of 3.0 will support the dos.library SetVBuf() function. From V36 to V39 this call did nothing. It will now properly change buffering modes, sizes, etc. For programs using Dos buffered I/O calls (FGetC, FPutC, FRead, etc.) this can make a major improvement in I/O performance if the buffer size is increased (and if the buffering mode is changed when writing to BUF_FULL).

For additional information on OS changes, take particular note of all BUGS and NOTES entries in the system autodocs.

Taking Advantage of New Features

By simply using system features and functions which were introduced in Release 2, you can write adaptable software that can automatically grow and support many new system software and hardware capabilities.

Add a bit of conditional code to treat color guns (R, G, and B) as at least 8-bits each, and to conditionally use the new 32-bit V39 color setting and getting functions, and your application can provide full AA palette support.

Here are some strategies for adapting to new system capabilities.

- ❑ Use the `asl.library` requesters if available. They are localized, they cut down on the amount of code you have to maintain, and they grow with the system. 2.0 Workbench licensees may get a free amendment to distribute the 2.1 `asl.library` which contains the screen mode requester.
- ❑ Adapt to previously impossible deeper modes. Properly written software can use the calling syntax of the current graphics and Intuition library functions to open the screen modes with larger numbers of bitplanes. These modes are simply deeper bitplane versions of existing modes: Lores 6,7, and 8 bits; Hires 5,6,7, and 8 bits; SuperHires 3,4,5,6,7, and 8 bits; and VGA 3,4,5,6,7, and 8 bits.

Note that the display database must be checked to see if the user's machine supports the desired mode and number of bitplanes.

Intuition's screen functions can be used to open these previously impossible screens, but programs cannot rely on Intuition to throw out all "bad" possibilities. For example, V36 Intuition will open an 8-bitplane Lores screen, but this is a waste of memory if the new chipset is not present since the 3 extra bitplanes will simply go unused. Programs will therefore need to explicitly check the display database to find out what the chips support. In V39, Intuition will check the chips for you and fail to open a screen which is "too deep." There is a new `SA_ErrorCode` value, `OSERR_TOODEEP`.

Note on Mode IDs. The currently defined display database mode IDs have all been moved from `<graphics/displayinfo.h>` to the new include file `<graphics/modeid.h>`. Do not attempt to create your own display mode IDs by OR'ing together monitors and modes unless the include file explicitly allows it. Do not try to draw conclusions about display characteristics by interpreting bits in the display mode ID. Instead, pass the mode ID to the display database functions to discover the display characteristics (see the example below and other examples in *Amiga Mail*, *DevCon Notes*, *Amiga Rom Kernel Reference Manual: Libraries*, and the new screen mode requester example).

The complete example listed below will check the display database entry for a LORES screen and open a screen with the maximum allowable number of bitplanes (assuming there is enough Chip memory). On an Amiga with any chipset up to and including ECS, this code will open a 5-bitplane screen. On a machine with the AA chipset, the screen will be 8 bits deep. The code will then draw one vertical line for each available color using Intuition's preferred palette (which can be set by the user with Preferences). The example requires V36 or a later version of the OS.

```

/* maxdepthlores.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/displayinfo.h>
#include <dos/dos.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/dos_protos.h>

#include <stdio.h>
#include <stdlib.h>

struct Library *IntuitionBase;
struct Library *GfxBase;

void Quit(char *whytext, LONG failcode)
{
    if(*whytext)
        printf("%s", whytext);

    if (IntuitionBase)
        CloseLibrary(IntuitionBase);

    if (GfxBase)
        CloseLibrary(GfxBase);

    exit(failcode);
}

void main(void)
{
    ULONG modeID = LORES_KEY;
    DisplayInfoHandle displayhandle;
    struct DimensionInfo dimensioninfo;

    UWORD maxdepth, maxcolors;
    ULONG soerror = NULL, colornum;
    struct Screen *screen;

    if ((GfxBase = OpenLibrary("graphics.library", 36)) == NULL)
        Quit("graphics.library is too old <V36", RETURN_FAIL);

    if ((IntuitionBase = OpenLibrary("intuition.library", 36)) == NULL)
        Quit("intuition.library is too old <V36", RETURN_FAIL);

    if ((displayhandle = FindDisplayInfo(modeID)) == NULL)
        Quit("modeID not found in display database", RETURN_FAIL);

    if (GetDisplayInfoData(displayhandle, (UBYTE *)&dimensioninfo,
        sizeof(struct DimensionInfo), DTAG_DIMS, NULL) == 0)
        Quit("mode dimension info not available", RETURN_FAIL);

    maxdepth = dimensioninfo.MaxDepth;
    printf("dimensioninfo.MaxDepth=%d", (int) maxdepth);

    if (screen = OpenScreenTags(NULL, SA_DisplayID, modeID,
        SA_Depth, (UBYTE) maxdepth,
        SA_Title, "MaxDepth LORES",
        SA_ErrorCode, &soerror,
        SA_FullPalette, TRUE,
        TAG_END))

```



```

{
/* Zowie! we actually got the screen open! now let's try drawing into it. */
maxcolors=1<<maxdepth;
printf("maxcolors=%d", (int) maxcolors);

for(colornum=0;colornum<maxcolors;++colornum)
{
SetAPen(&(screen->RastPort),colornum);
Move(&(screen->RastPort),colornum,screen->BarHeight + 2);
Draw(&(screen->RastPort),colornum,screen->Height - 1);
}

Delay(TICKS_PER_SECOND * 6);
CloseScreen(screen);
}
else
{
/* Hmmm. Couldn't open the screen. maybe not enough CHIP RAM?
* Maybe not enough chips! ;-)
*/
switch(soerror)
{
case OSERR_NOCHIPS:
Quit("Bummer! You need new chips dude!",RETURN_FAIL);
break;
case OSERR_UNKNOWNMODE:
Quit("Bummer! Unknown screen mode.",RETURN_FAIL);
break;
case OSERR_NOCHIPMEM:
Quit("Not enough CHIP memory.",RETURN_FAIL);
break;
case OSERR_NOMEM:
Quit("Not enough FAST memory.",RETURN_FAIL);
break;
default:
printf("soerror=%d",soerror);
Quit("Screen opening error.",RETURN_FAIL);
break;
}
Quit("Couldn't open screen.",RETURN_FAIL);
}
Quit("",RETURN_OK); /* clean up and exit */
}

```

☐ **Open on Workbench.** One of the easiest ways to help a given program peacefully coexist with new hardware is to allow it to open on the Workbench screen (or on any public screen). Software written this way should be robust enough that it can find out from the system any information it might need about the display environment, and, if it understands the kind of screen that it's on, use the appropriate graphics.library calls to manipulate its windows.

Here's some example code that determines the depth of the default public screen and opens a window on it. If it were part of a real application, this code would tell how many colors the screen has.

Add some conditional V39 palette sharing code and you can even have your own color registers to use, if available, or closest matching register to share.

```
/* depthawarevisitor.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/displayinfo.h>
#include <dos/dos.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/dos_protos.h>

#include <stdio.h>
#include <stdlib.h>

struct Library *IntuitionBase;
struct Library *GfxBase;
struct Screen *screen = NULL;

void Quit(char *whytext, LONG failcode)
{
    if(*whytext)
        printf("%s", whytext);

    if (screen)
        UnlockPubScreen(NULL, screen);

    if (IntuitionBase)
        CloseLibrary(IntuitionBase);

    if (GfxBase)
        CloseLibrary(GfxBase);

    exit(failcode);
}

void main(void)
{
    struct Screen *screen;
    struct DrawInfo *drawinfo;
    struct Window *window;
    UWORD depth;

    if ((GfxBase = OpenLibrary("graphics.library", 36)) == NULL)
        Quit("graphics.library is too old <V36", RETURN_FAIL);

    if ((IntuitionBase = OpenLibrary("intuition.library", 36)) == NULL)
        Quit("intuition.library is too old <V36", RETURN_FAIL);

    if (!(screen = LockPubScreen(NULL)))
        Quit("Can't lock default public screen", RETURN_FAIL);

    /* Here's where we'll ask Intuition about the screen. */
    if ((drawinfo = GetScreenDrawInfo(screen)) == NULL)
        Quit("Can't get DrawInfo", RETURN_FAIL);
    depth = drawinfo->dri_Depth;

    /* Because Intuition allocates the DrawInfo structure, we have to tell it when
     * we're done, to get the memory back.
     */
    FreeScreenDrawInfo(screen, drawinfo);
}
```

```

/* This next line takes advantage of the stack-based amiga.lib version of
OpenWindowTags().
*/
if (window = OpenWindowTags(NULL, WA_PubScreen ,screen,
    WA_Left, 0,
    WA_Width, screen->Width,
    WA_Top, screen->BarHeight,
    WA_Height, screen->Height - screen->BarHeight,
    WA_Flags, WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE|
    ACTIVATE|SIMPLE_REFRESH|NOCAREREFRESH,
    WA_Title, "Big Visitor",
    TAG_END))
{
    printf("depth=%d",depth);

    /* All our window event handling might go here */

    Delay(TICKS_PER_SECOND * 10);

    /* Of course, some other program might come along and change the attributes of
    * the screen that we read from DrawInfo, but that's a mean thing to do to a
    * public screen, so let's hope it doesn't happen.
    */

    CloseWindow(window);
}

Quit("",RETURN_OK); /* clean up (close/unlock) and exit */
}

```

- ❑ If necessary, use tag-extended older structures with 1.3 functions. If you are not yet able to drop support for 1.3 systems, use tag-extended structures such as ExtNewScreen and ExtNewWindow to provide V37 and V39/AA enhanced capabilities with 1.3-compatible code. See the Intuition chapters of the *Amiga ROM Kernel Reference Manual: Libraries* for examples that use the tag-extended structures. An example of such coding is the NewIFF39 code.

New AA Display Modes (in addition to modes supported by ECS)

Display Mode	Planes	Colors	Bandwidth (see note 1)
LORES (320 x 200 NTSC or 320 x 256 PAL)	6	64	1x
	7	128	1x
	8	256	1x
	8 HAM	256,000+ (see note 2)	1x
HIRES (640 x 200 NTSC or 640 x 256 PAL)	5	32	2x
	6 EHB	64 (see note 3)	2x
	6 HAM	4096 (see note 4)	2x
	6	64	2x
	7	128	2x
	8	256	2x
	8 HAM	256,000+ (see note 2)	2x
SUPERHIRES (1280 x 200 NTSC or 1280 x 256 PAL)	1	2 (see note 5)	1x
	2	4 (see note 5)	1x
	3	8	2x
	4	16	2x
	5	32	4x
	6 EHB	64 (see note 3)	4x
	6 HAM	4096 (see note 4)	4x
	6	64	4x
	7	128	4x
	8	256	4x
	8 HAM	256,000+ (see note 2)	4x
MULTISCAN (160, 320, 640 x 480 non-interlaced)	1	2 (see note 5)	1x
	2	4 (see note 5)	1x
	3	8	2x
	4	16	2x
	5	32	4x
	6 EHB	64 (see note 3)	4x
	6 HAM	4096 (see note 4)	4x
	6	64	4x
	7	128	4x
	8	256	4x
	8 HAM	256,000+ (see note 2)	4x

New AA Display Modes (in addition to modes supported by ECS)

Display Mode	Planes	Colors	Bandwidth (see note 1)
SUPER72 (71 Hz refresh and 23.21 kHz scan rate; 200, 400, 800 x 300 non-interlaced or 200, 400, 800 x 600 interlaced)	1	2 (see note 5)	1x
	2	4 (see note 5)	1x
	3	8	2x
	4	16	2x
	5	32	4x
	6 EHB	64 (see note 3)	4x
	6 HAM	4096 (see note 4)	4x
	6	64	4x
	7	128	4x
EURO72 (69 Hz refresh and 29.32 kHz scan rate; 160, 320, 640 x 480 non-interlaced)	8	256	4x
	8 HAM	256,000+ (see note 2)	4x
	1	2 (see note 5)	1x
	2	4 (see note 5)	1x
	3	8	2x
	4	16	2x
	5	32	4x
	6 EHB	64 (see note 3)	4x
	6 HAM	4096 (see note 4)	4x
DOUBLENTSC (58 Hz refresh and 27.66 kHz scan rate; 160, 320, 640 x 280 scan-doubled or 160, 320, 640 x 480 non-interlaced or 160, 320, 640 x 800 interlaced)	6	64	4x
	7	128	4x
	8	256	4x
	8 HAM	256,000+ (see note 2)	4x
	1	2 (see note 5)	1x
	2	4 (see note 5)	1x
	3	8	2x
	4	16	2x
	5	32	4x
	6 EHB	64 (see note 3)	4x
	6 HAM	4096 (see note 4)	4x
	6	64	4x
	7	128	4x
	8	256	4x
	8 HAM	256,000+ (see note 2)	4x

New AA Display Modes (in addition to modes supported by ECS)

Display Mode	Planes	Colors	Bandwidth (see note 1)
DOUBLEPAL	1	2 (see note 5)	1x
(58Hz refresh and	2	4 (see note 5)	1x
27.66 kHz scan rate;	3	8	2x
160, 320, 640 x 256	4	16	2x
scan-doubled or	5	32	4x
160, 320, 640 x 512	6 EHB	64 (see note 3)	4x
non-interlaced or	6 HAM	4096 (see note 4)	4x
160, 320, 640 x 1024	6	64	4x
interlaced)	7	128	4x
	8	256	4x
	8 HAM	256,000+ (see note 2)	4x

Notes

1 - The bandwidth number describes the relative amount of fetch bandwidth required by a particular screen mode. For example, a 5-bit deep MultiScan screen requires 4x fetch bandwidth while a 1-bit MultiScan screen requires only 1x. This means the hardware has to move data 4 times faster on the deeper screen. To be able to move data at these higher rates, the higher bandwidth modes require data to be properly aligned in Chip memory that is fast enough to support the bandwidth. Specifically, bandwidth factors of 1x require data to be on 16-bit boundaries, factors of 2x require 32-bit boundaries, and factors of 4x require 64-bit boundaries.

Restrictions like these are the best reason to use the system allocation functions whenever data is being prepared for the custom hardware. It is not guaranteed that all machines that have the new chip set will also have memory fast enough for the 4x modes. Therefore, the only way to know whether or not the machine will support the mode you want is to check the display database.

2 - New 8-bit HAM mode uses the upper 6 bits as an offset into a table of 64 24-bit base color registers and the lower 2 bits for modify mode control. This mode could conceivably allow simultaneous display of more than 256,000 colors (up to 16.8 million, presuming a monitor and screen mode with enough pixels). Please note that while the register planes and control planes are internally reversed in 8-bit HAM (the modify control bits are the two LSBs instead of the two MSBs), programs using graphics library and Intuition will not have to deal with this reversal, as it will be handled automatically for them.

3 - This is like the original EHB mode, but in new resolutions. It uses 5 bits as an offset into a table of 32 color registers plus a sixth bit to yield an additional 32 colors that are 1/2 as bright.

4 - This is like the original 6-bit HAM mode, but in new resolutions. It uses the lower 4 bits as an offset into a table of 16 color registers and the upper 2 bits for modify mode control. This mode allows simultaneous display of 4096 colors.

5 - These modes are similar to the VGA and SuperHires modes of the ECS chipset except that they are not restricted to a nonstandard 64-color palette.





ARexx

by Michael Sinz, Christian Ludwig & Jerry Hartlzer

SimpleRexx: A Simple ARexx Interface

by Michael Sinz

The ability to handle scripts is a powerful feature for any application. Whenever users have a repetitive, well-defined job to do with application software, script capabilities allow them to perform the job automatically without any user intervention. For example, many communication programs allow the user to set up a script which will automatically dial up a host system, login to it, and check for messages to download. To encourage the development of more applications which have this powerful feature, Commodore has added ARexx to V2.0 of the Amiga system software.

What Is ARexx?

ARexx is the Amiga implementation of the REXX language. Like BASIC, ARexx is an interpreted language. ARexx can be used alone for almost any programming job, but the real power of ARexx is unleashed when it is used with applications as a system-level scripting language. With the addition of its own ARexx port and a small amount of code, an application can gain all the power scripting capabilities offer. There are other benefits as well. The burden of writing code to handle scripts is reduced significantly. Also, ARexx helps provide a consistent interface to the user. Without it, every developer who implements scripts must invent their own scripting language, forcing the user to learn several different scripting languages rather than just one.

Perhaps best of all is that applications which support ARexx can "talk" to each other by sending commands and sharing data - even if they are from different vendors. For instance, a communications program with an ARexx port could be set up to automatically download data from a financial bulletin board and then pass the data to a spreadsheet application to create a graph or financial model. The ability of applications to talk to one another on a multitasking computer is known as Inter-process Communication (IPC).

Because it interprets scripts and passes strings, ARexx has been optimized for string processing. Almost all ARexx interactions involve passing a string; numbers are even passed as ASCII representations in most situations.

ARexx has enhanced abilities to send and receive control messages to and from many sources. An application that supports ARexx can send and receive these control messages. The messages contain text strings that are either interpreted directly by ARexx itself or that are passed on as commands to be processed by the destination application.

ARexx also supplies methods for applications to send and receive data through an ARexx port. The data can be sent in a message or via the ARexx RVI (Rexx Variable Interface). In either case, data can be transferred to and from ARexx. A complete ARexx supporting application would need to be able to send data to a requesting ARexx program/script and get data from that program.

The SimpleRexx Routines

Adding an ARexx interface to an application can be difficult, especially when working with it for the first time. SimpleRexx was written to serve not only as an example of how to implement an ARexx interface but also as a "wrapper" to be incorporated into other code. It contains routines to take care of the lower level ARexx work. It can be used to add minimal ARexx support to many applications in a backwards-compatible fashion. In other words, an application that uses SimpleRexx will still work on systems without ARexx (but won't be able to execute ARexx scripts).

The code below consists of SimpleRexx.c, the wrapper code that makes up the ARexx interface, and an example, SimpleRexxExample.c, which illustrates the use of the wrapper. The test.rexx script is an example ARexx script to execute while SimpleRexxExample is running. It will send commands to SimpleRexxExample in order to control it. The test.results file shows the output of test.rexx.

In addition to the code examples listed here, you will need to install the rexxsyslib.library and rexxsupport.library in the libs: directory of the target machine. Don't forget to give the rexxmast command either in your startup-sequence or from the CLI in order to start ARexx.

Overview of Functions

The source to SimpleRexx is a single file, SimpleRexx.c. The header file, SimpleRexx.h, contains the type definitions and prototypes for the functions.

The SimpleRexx functions are used as follows:

```
rexx_context=InitARexx(AppBaseName, Extension)
```

This allocates and initializes a SimpleRexx ARExxContext structure. This context is much like a file handle in that it will be used in all other calls that make use of this SimpleRexx context. Since all SimpleRexx calls correctly check the rexx_context before doing work, you do not need to check the return result of this call. If ARExx is not available on your system, SimpleRexx won't do anything.

```
port_name=ARExxName(rexx_context)
```

This function returns a pointer to the name of the ARExx port's context. The name is based on the AppBaseName plus an invocation number allowing multiple copies of an application to run at the same time. If you have no ARExx port, it returns NULL.

```
sigmask=ARExxSignal(rexx_context)
```

This function returns the signal bit mask of your context's ARExx port. This should be combined with other signal masks to produce the argument to the Wait() call. This returns NULL if there is no signal mask.

```
rmsg=GetARExxMsg(rexx_context)
```

This function returns the next ARExx message that is waiting. rmsg=NULL if there is no message or ARExx is not around. rmsg=REXX_RETURN_ERROR if a message sent to ARExx via SendARExxMsg() returns an error.

```
ReplyARExxMsg(rexx_context, rmsg, result, error)
```

This function sends back the ARExx message received via GetARExxMsg(). The result field is a pointer to a result string that is returned via OPTIONS RESULTS in the RESULT ARExx variable. If you have no result, set this to NULL. Error is the error severity level. If this is 0, the result string will be returned. If this is non-zero, the error level will be returned in RC.

```
worked=SendARExxMsg(rexx_context, string, StringFileFlag)
```

This function sends a string to ARExx. It sets the default host to the context and sets the RXFB_STRING bit in the message if the StringFileFlag is set. This routine returns FALSE if the message was not sent.

```
worked=SetARExxLastError(rexx_context, rmsg, ErrorString)
```

This function uses the RVI (Rexx Variable Interface) to set a variable named

<AppBaseName>.LASTERROR to the ErrorString. This is where the error message should go if there is an error. This function returns FALSE if this fails. You must call this routine after a GetARexxMsg() and before ReplyARexxMsg().

FreeARexx(rexx_context)

This closes a SimpleRexx context received from InitARexx().

The Future of ARexx

ARexx is much more than a system-level scripting language. Among other things, ARexx can be used like glue to put together a set of application modules. This frees the programmer from worrying about data sharing between modules so that more effort can be put into refining the code modules themselves. It also allows a user who understands ARexx to add their own refinements creating a truly custom environment. This customizability is also ideal for VARs or dealers who want to sell vertical market solutions.

```

$ Makefile for SimpleRexx and SimpleRexxExample
$
CFLAGS= -b1 -cflat -d0 -ma0 -rx1 -v -w
HEAD= SimpleRexx.h
CODE= SimpleRexx.c SimpleRexxExample.c
OBJ= SimpleRexx.o SimpleRexxExample.o
LIB= LIB:lib rexware.o Lib:amiga.lib
.C.o:
    RLC $(CFLAGS) $*
SimpleRexxExample: $(OBJ) $(LIBS)
    Link FROM Lib:c.o $(OBJ) TO SimpleRexxExample LIB $(LIBS) SD SC MD
SimpleRexx.o: SimpleRexx.c SimpleRexx.h
SimpleRexxExample.o: SimpleRexxExample.c SimpleRexx.h
/*
 * file: simpleRexx.h
 *
 * Simple Abrex interface...
 *
 * This is a very "simple" interface...
 */
#ifdef SIMPLE_REXX_H
#define SIMPLE_REXX_H
#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/ports.h>
#include <exec/storage.h>
#include <exec/raslib.h>
/*
 * This is the handle that SimpleRexx will give you
 * when you initialise an Abrex port...
 *
 * The conditional below is used to skip this if we have
 * defined it earlier...
 */
#ifdef ABREXCONTEXT
typedef void *ABREXCONTEXT;
#endif /* ABREXCONTEXT */
/*
 * The value of REXXMSG (from GetAbrexMsg) if there was an error returned
 */
#define REXX_RETURN_ERROR ((struct REXXMSG *)-1L)
/*
 * This function closes down the Abrex context that was opened
 * with InitAbrex...
 */
void FreeAbrex(ABREXCONTEXT);
/*
 * This routine initialises an Abrex port for your process
 * This should only be done once per process. You must call it
 * with a valid application name and you must use the handle it
 * returns in all other calls...

```

```

 * NOTE: The AppName should not have spaces in it...
 * Example AppNames: "MyWord" or "FastCalc" etc...
 * The name "MUST" be less than 16 characters...
 * If it is not, it will be trimmed...
 * The name will also be UPPER-CASED...
 *
 * NOTE: The Default file name extension, if NULL will be
 * "rexx" (the "." is automatic)
 */
ABREXCONTEXT InitAbrex(char *,char *);
/*
 * This function returns the port name of your Abrex port.
 * It will return NULL if there is no Abrex port...
 *
 * This string is "READ ONLY" You "MUST NOT" modify it...
 */
char *AbrexName(ABREXCONTEXT);
/*
 * This function returns the signal mask that the REXX port is
 * using. It returns NULL if there is no signal...
 *
 * Use this signal bit in your Wait() loop...
 */
ULONG AbrexSignal(ABREXCONTEXT);
/*
 * This function returns a structure that contains the commands sent from
 * Abrex... You will need to parse it and return the structure back
 * so that the memory can be freed...
 *
 * This returns NULL if there was no message...
 */
struct REXXMSG *GetAbrexMsg(ABREXCONTEXT);
/*
 * Use this to return a Abrex message...
 *
 * If you wish to return something, it must be in the REXXMSG.
 * If you wish to return an error, it must be in the error.
 */
void ReplyAbrexMsg(ABREXCONTEXT,struct REXXMSG *,char *,LONG);
/*
 * This function will send a string to Abrex...
 *
 * The default host port will be that of your task...
 *
 * If you set stringFile to TRUE, it will set that bit for the message...
 *
 * Returns TRUE if it send the message, FALSE if it did not...
 */
short SendAbrexMsg(ABREXCONTEXT,char *,short);
/*
 * This function will set an error string for the Abrex
 * application in the variable defined as <appName>:LASTERROR
 *
 * Note that this can only happen if there is an Abrex message...
 *
 * This returns TRUE if it worked, FALSE if it did not...
 */
short SetAbrexLastError(ABREXCONTEXT,struct REXXMSG *,char *);
#endif /* SIMPLE_REXX_H */
/*
 * File: simpleRexx.c
 *
 * Simple Abrex interface...

```



```

* If you wish to return something, it must be in the RString.
* If you wish to return an error, it must be in the Error.
* If there is an error, the RString is ignored.
void ReplyArenxMsg(ArenxContext ArenxContext, struct RMsg *rmsg,
    char *RString, LONG Error)
{
    if (ArenxContext) if (rmsg) if (rmsg->RXX_RETURN_ERROR)
    {
        rmsg->RXX_Result2=0;
        if (!rmsg->RXX_Result1-Error)
        {
            /*
            * If you did not have an error we return the string
            */
            if (rmsg->RXX_Action & (1L < RXX_RESULT)) if (RString)
            {
                rmsg->RXX_Result2=(LONG)CreateArgString(RString,
                    (LONG)strlen(RString));
            }
        }
        /*
        * Reply the message to Arenx...
        */
        ReplyMsg((struct Message *)rmsg);
    }
}

/*
* This function will set an error string for the Arenx
* application in the variable defined as <apname>.LASTERROR
*
* Note that this can only happen if there is an Arenx message...
*
* This returns TRUE if it worked, FALSE if it did not...
*/
short SetArenxLastError(ArenxContext ArenxContext, struct RMsg *rmsg,
    char *ErrorString)
{
    register short OKFlag=FALSE;

    if (ArenxContext) if (rmsg) if (CheckRxxMsg(rmsg))
    {
        /*
        * Note that SetRxxVar() has more than just a TRUE/FALSE
        * return code, but for this "basic" case, we just care if
        * it works or not.
        */
        if (!SetRxxVar(rmsg, ArenxContext->ErrorMessage, ErrorString,
            (long)strlen(ErrorString)))
        {
            OKFlag=TRUE;
        }
        return(OKFlag);
    }

    /*
    * This function will send a string to Arenx...
    *
    * The default host port will be that of your task...
    *
    * If you set StringFile to TRUE, it will set that bit for the message...
    *
    * Returns TRUE if it send the message, FALSE if it did not...
    */
    short SendArenxMsg(ArenxContext ArenxContext, char *RString,
        short StringFile)
    {
        register struct MsgPort *RxxPort;
        register struct RMsg *rmsg;
        register short flag=FALSE;

```

```

    if (ArenxContext) if (RString)
    {
        if (rmsg->CreateRxxMsg(ArenxContext->ArenxPort,
            ArenxContext->Extension,
            ArenxContext->PortName))
        {
            rmsg->RXX_Action=RXXCOMM | (StringFile ?
                (1L < RXX_STRING):0);
            if (rmsg->RXX_Args[0]-CreateArgString(RString,
                (LONG)strlen(RString)))
            {
                /*
                * We need to find the RxxPort and this needs
                * to be done in a Forbid()
                */
                Forbid();
                if (RxxPort=FindPort(RXSDIR))
                {
                    /*
                    * We found the port, so put the
                    * message to Arenx...
                    */
                    PutMsg(RxxPort, (struct Message *)rmsg);
                    ArenxContext->Outstanding+=1;
                    flag=TRUE;
                }
                else
                {
                    /*
                    * No port, so clean up...
                    */
                    DeleteArgString(rmsg->RXX_Args[0]);
                    DeleteRxxMsg(rmsg);
                }
                Permit();
            }
            else DeleteRxxMsg(rmsg);
        }
        return(flag);
    }

    /*
    * This function closes down the Arenx context that was opened
    * with InitArenx...
    */
    void FreeArenx(ArenxContext ArenxContext)
    {
        register struct RxxMsg *rmsg;
        if (ArenxContext)
        {
            /*
            * Clear port name so it can't be found...
            */
            ArenxContext->PortName[0]='0';

            /*
            * Clean out any outstanding messages we had sent out...
            */
            while (ArenxContext->Outstanding)
            {
                WaitPort(ArenxContext->ArenxPort);
                while (rmsg=GetArenxMsg(ArenxContext))
                {
                    if (rmsg->RXX_RETURN_ERROR)
                    {
                        /*
                        * Any messages that come now are blown
                        * away...
                        */
                        SetArenxLastError(ArenxContext, rmsg,
                            "99: Port Closed!");

```



```

ReplyARexMsg(RexContext, rmsg,
             NULL, 100);
)
)
/*
 * Clean up the port and delete it...
 */
if (RexContext->ARexPort)
{
    while (rmsg->GetARexMsg(RexContext))
    {
        /*
         * Any messages that still are coming in are
         * "dead" We just set the LASTERROR and
         * reply an error of 100...
         */
        SetARexLastError(RexContext, rmsg,
                        "99; Port Closed!");
        ReplyARexMsg(RexContext, rmsg, NULL, 100);
    }
    DeletePort(RexContext->ARexPort);
}
/*
 * Make sure we close the library...
 */
if (RexContext->RexSysBase)
{
    CloseLibrary(RexContext->RexSysBase);
}
/*
 * Free the memory of the RexContext
 */
FreeMem(RexContext, sizeof(struct ARexContext));
}
}
/*
 * This routine initializes an ARex port for your process
 * This should only be done once per process. You must call it
 * with a valid application name and you must use the handle it
 * returns in all other calls...
 *
 * NOTE: The AppName should not have spaces in it...
 * Example AppNames: "Myword" or "FastCalc" etc...
 * The name "must" be less than 16 characters...
 * If it is not, it will be trimmed...
 * The name will also be UPPER-CASED...
 *
 * NOTE: The Default file name extension, if NULL will be
 * "rexx" (the "." is automatic)
 */
AREXCONTEXT InitARex(char *AppName, char *Extension)
{
    Register ALEXCONTEXT RexContext=NULL;
    register short loop;
    register short count;
    register char *tmp;

    if (RexContext=AllocMem(sizeof(struct ARexContext),
                          MEM_PUBLIC|MEM_CLEAR))
    {
        if (RexContext->RexSysBase=OpenLibrary("rexxsyslib.library",
                                              NULL))
        {
            /*
             * Set up the extension...
             */
            if ((Extension) Extension="rexx";
                tmp=RexContext->Extension;
                for (loop=0; (loop<7)&&(Extension[loop]); loop++)

```

```

{
    *tmp++=Extension[loop];
}
*tmp='0';
/*
 * Set up a port name...
 */
tmp=RexContext->PortName;
for (loop=0; (loop<16)&&(AppName[loop]); loop++)
{
    *tmp++=toupper(AppName[loop]);
}
*tmp='0';
/*
 * Set up the last error RVI name...
 */
/* This is <appName>.LASTERROR
 */
strcpy(RexContext->ErrorMessage, RexContext->PortName);
strcat(RexContext->ErrorMessage, ".LASTERROR");
/* We need to make a unique port name... */
forbid();
for (count=1, RexContext->ARexPort=(VOID *)1;
    RexContext->ARexPort/count++);
{
    stc d(tmp, count);
    RexContext->ARexPort=
        FindPort(RexContext->PortName);
}
RexContext->ARexPort=CreatePort(
    RexContext->PortName, NULL);
Permit();
if ( ((RexContext->RexSysBase))
    || ((RexContext->ARexPort)) )
{
    FreeARex(RexContext);
    RexContext=NULL;
}
return(RexContext);
}
/*
 * File: SimpleRexExample.c
 * This is an example of how to use the SimpleRex code.
 */
#include <exec/types.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <intuition/intuition.h>
#include <proto/exec.h>
#include <proto/intuition.h>
#include <rexx/storage.h>
#include <rexx/raslib.h>
#include <stdio.h>
#include <string.h>
#include "simpleRex.h"
/*

```

```

/* Lattice control-c stop...
*/
int CHKABORT(void) { return(0); } /* Disable Lattice CTRL/C handling */
int CHKABORT(void) { return(0); } /* really */

/* The strings in this program
*/
char *strings[] = {
    "50: Window already open", /* STR_ID_WINDOW_OPEN */
    "101: Window did not open", /* STR_ID_WINDOW_ERROR */
    "50: No Window", /* STR_ID_WINDOW_NONE */
    "60: Argument error to WINDOW command", /* STR_ID_WINDOW_ARG */
    "100: Unknown command", /* STR_ID_COMMAND_ERROR */
    "Alexx port name: 66", /* STR_ID_PORT_NAME */
    "No Alexx on this system.", /* STR_ID_NO_ALEXX */
    "SimplexExample Window", /* STR_ID_WINDOW_TITLE */
};

#define STR_ID_WINDOW_OPEN 0
#define STR_ID_WINDOW_ERROR 1
#define STR_ID_WINDOW_NONE 2
#define STR_ID_WINDOW_ARG 3
#define STR_ID_COMMAND_ERROR 4
#define STR_ID_PORT_NAME 5
#define STR_ID_NO_ALEXX 6
#define STR_ID_WINDOW_TITLE 7

/* NewWindow structure...
*/
static struct NewWindow new-
{
    97,47,299,44,-1,-1,
    CLOSEBUTTON,
    WINDOWSTYLE|WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE|
    STYLE_REFRESH|DCOUREREFRESH,
    NULL, NULL,
    NULL, NULL,290,40,-1,-1,WSCHSCREEN
};

/* A "VERY" simple and simple-minded example of using the simplex.c code.
*
* This program, when run, will print out the name of the Alexx port it
* opens. Use that port to tell it to show the window. You can also
* use the Alexx port to HIDE the window, to READTITLE the window's
* titlebar, and QUIT the program. You can also quit the program by
* pressing the close gadget in the window while the window is up.
*
* Note: You will want to RUN this program or have another shell available such
* that you can still have access to Alexx...
*/
void main(int argc, char *argv[])
{
    short loopflag=TRUE;
    ALEXXCONTEXT Alexxstuff;
    struct Window *win=NULL;
    ULONG signals;

    if (IntuitionBase->(struct IntuitionBase *)
        OpenLibrary("intuition.library",NULL))
    {
        /*
        * Note that Simplex is set up such that you do not
        * need to check for an error to initialise your REXX port
        * This is so your application could run without REXX...
        */
        Alexxstuff-InitAlexx("Example","test");

        if (argc)
        {

```

```

if (Alexxstuff) printf(strings(STR_ID_PORT_NAME),
    AlexxName(Alexxstuff));
else printf(strings(STR_ID_NO_ALEXX));
}

while (loopflag)
{
    signals=AlexxSignal(Alexxstuff);
    if (win) signals|=1L << (win->UserPort->mp_sigbit);

    if (signals)
    {
        struct REXXMsg *rmsg;
        struct IntuiMessage *msg;
        signals=Wait(signals);

        /* Process the Alexx messages...
        */
        while (rmsg=GetAREXXMsg(Alexxstuff))
        {
            char cbuf[24];
            char *nextchar;
            char *error=NULL;
            char *result=NULL;
            long errlevel=0;

            nextchar=strcpy(cbuf, rmsg);
            cbuf[24]=0;
            if (*nextchar) nextchar++;

            if (!strcmp("WINDOW",cbuf))
            {
                if (!strcmp("OPEN",nextchar))
                {
                    if (win)
                    {
                        error=strings(STR_ID_WINDOW_OPEN);
                        errlevel=5;
                    }
                    else
                    {
                        nw_title=strings(STR_ID_WINDOW_TITLE);
                        if (!!(win=OpenWindow(&nw)))
                        {
                            error=strings(STR_ID_WINDOW_ERROR);
                            errlevel=30;
                        }
                    }
                }
                else if (!strcmp("CLOSE",nextchar))
                {
                    if (win)
                    {
                        CloseWindow(win);
                        win=NULL;
                    }
                    else
                    {
                        error=strings(STR_ID_WINDOW_NONE);
                        errlevel=5;
                    }
                }
                else
                {
                    error=strings(STR_ID_WINDOW_ARG);
                    errlevel=20;
                }
            }
            else if (!strcmp("READTITLE",cbuf))
            {
                if (win)
            }

```

```

    result-win->Title;
    }
    else
    {
        error-strings[STR_ID_WINDOW_NONE];
        errlevel=5;
    }
    }
    else if (!strcmp("QUIT",cbuf))
    {
        loopflag=FALSE;
    }
    else
    {
        error-strings[STR_ID_COMMAND_ERROR];
        errlevel=20;
    }
    }
    if (error)
    {
        SetLastError(Rexxstuff,rmag,error);
        ReplyRexxMsg(Rexxstuff,rmag,result,errlevel);
    }
    /* * If we have a window, process those messages
    */
    if (win) while (msg=(struct IntuiMessage *)
        GetMsg(win->UserPort))
    {
        if (msg->Class==CLOSEWINDOW)
        {
            /*
            * Quit if the close gadget...
            */
            loopflag=FALSE;
        }
        ReplyMsg((struct Message *)msg);
    }
    else loopflag=FALSE;
    }
    if (win) CloseWindow(win);
    FreeRexx(Rexxstuff);
    CloseLibrary((struct Library *)IntuitionBase);
}

/*
* File: test.rexx
*
* SimpleRexx test...
*
* You need to run the SimpleRexxExample first...
*/
Options FailAt 100
Options Results

/*
* Try to read the window title bar
*/
Address EXAMPLE1 ReadTitle
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window title is 'Result
/*
* Bad WINDOW command...

```

```

/*
Address EXAMPLE1 "Window Display"
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window is now open'
/*
* Open the window
*/
Address EXAMPLE1 "Window Open"
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window is now open'
/*
* Open the window
*/
Address EXAMPLE1 "Window Open"
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window is now open'
/*
* Try to read the window title bar
*/
Address EXAMPLE1 ReadTitle
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window title is 'Result
/*
* Hide the window
*/
Address EXAMPLE1 "Window Close"
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window is now closed'
/*
* Try to hide the window again
*/
Address EXAMPLE1 "Window Close"
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'Window is now closed'
/*
* Send a command that does not exist
*/
Address EXAMPLE1 Junk
if rc > 0 then say 'Error was 'EXAMPLE.LASTERROR
else say 'The command worked!!!'
/*
* Quit the program...
*/
Address EXAMPLE1 Quit

File: test.results
Error was 50: No Window
Error was 80: Argument error to WINDOW command
Window is now open
Error was 50: Window already open
Window title is SimpleRexxExample Window
Window is now closed
Error was 50: No Window
Error was 100: Unknown command

```

Using ASCII2AG.ttx

by Jerry Hartzler

When I started work for the CATS Department, my title was CD Developer. My main job responsibility was to put together release 2.0 of the CATS Developer CD. One new feature on this release is the inclusion of all of the Amiga Mail Volume II technical articles in AmigaGuide format (AmigaGuide being a HyperText tool by the CATS Department). Formatting all 44 ASCII articles by hand would have been a nightmare, so I decided to write an ARexx macro, ASCII2AG.ttx, to do most of the work for me.

Basically, ASCII2AG.ttx will convert a specially formatted ASCII document into a simple AmigaGuide database. All of document's sections and sub-sections are converted into AmigaGuide NODEs and Link Points. The final result will be a document with a master table of contents showing all of the sections as 'buttons'. The user can then click a button to view the associated section. There may also be buttons at the end of a section for that section's sub-sections. At the end of a sub-section may be buttons for that sub-section's sub-sub-sections and so on. Therefore, a User can read a document in whatever order desired and will no longer need to read an entire document to find what he/she is looking for.

The user does not need to know how to make an AmigaGuide database in order to run this ARexx macro, but it would certainly help. For more information see *AmigaGuide* by David Junod in the 1991 Devcon notes, and *AmigaGuide 101* in an upcoming issue of Amiga Mail.

Most of the commands in ASCII2AG.ttx are TurboText ARexx extension commands and therefore the script must be run under TurboText. TurboText is a text editor developed by Martin Taillefer and published by Oxxi, Inc. Every feature of TurboText can be controlled via ARexx commands, allowing the user the flexibility to customize and enhance the program.

Prepping the ASCII File

Before ASCII2AG.ttx is run, the ASCII file that is to become the AmigaGuide database must be prepared. The macro is not smart enough to know where the sections and sub-sections are in the document, so you must tell it. This is done by adding a 'Section Tag' in front of each section and sub-section heading.

Section Tag template:

<Section Number>@<NODE Name> <Section Header>

<Section Number>

The Section number is the priority number for that section. For example, all sections in a document are section number one (1@), all sub-sections are section number two (2@), all sub-sub-sections are number three (3@), and so on. Sections are limited to ten deep (0@ - 9@), section number zero being the main table of contents. The order of sections must be in the order that they will be read.

(0@) Table of Contents

(1@) Section

(2@) Sub-Section

(3@) Sub-Sub-Section

(1@) Section

(2@) Sub-Section

(2@) Sub-Section

etc...

Section tags must start in the first column of the section header line in order for ASCII2AG.ttx to recognize it.

A document should start with a 0@ section tag. For those familiar with AmigaGuide this will become the MAIN node. The 0@ section tag can be either the document's table of contents or introduction section. The ASCII2AG.ttx macro will add one automatically if it cannot find one and if the user requests it. Only one 0@ section tag can be in a document.

<NODE Name>

This is an optional parameter that allows the user to specify the NODE name. If a NODE name is not specified, ASCII2AG.ttx will supply one. NODE names must only be a single word and must immediately be after the @ (at) sign. Remember, so as not to confuse AmigaGuide, each NODE must be assigned a different name.

If a NODE name is assigned to section 0@ it will be ignored, and replaced by "MAIN". The NODE name will become the first word in the Section Header.

<Section Header>

The section header is the name or title of the section. ASCII2AG.ttx will make it the title parameter of a NODE, which is the text that is displayed in the title bar of the AmigaGuide window during the display of the NODE. It will also be the Label parameter, or the text within the button, of that NODE's Link Point.

A space must separate it from the NODE Name or the @ (at) sign.

If a Section Header is not supplied AmigaGuide will display the NODE Name in its window's title bar instead.

Example Script

As an example, here is an article on the Window menu option under Workbench 2.0.

THE WINDOW MENU	BEFORE FORMATTING
NEW DRAWER This menu item allows you to create a new drawer in the selected workbench window.	
OPEN PARENT Open Parent will open the selected window's parent window.	
CLOSE This item will close the selected window.	
UPDATE Update will clear the contents icon information of the selected window.	
SELECT CONTENTS This item will select the contents of the selected window.	
CLEAN UP This will clean up and display the selected window's icons in a more orderly fashion.	
SNAPSHOT Use one of the two sub-menu items below to save icon information.	
WINDOW This will save the selected window's position and size.	
ALL This will save the selected window's position and size, as well as, the positions of all of the window's icons.	
SHOW This item allows you to either show only icons or all the files in a selected window.	
ONLY ICONS This option will show only the files with associated icons.	
ALL FILES This option will show all the files. If the window is viewed by icon, all of the files without icons will have temporary icons attached to them.	
VIEW BY There are several ways you can view the contents of the selected window.	
ICON This option will display the contents as icons.	
NAME This option will display the contents as text in alphabetical order.	
DATE This option will display the contents as text in the order that they were created.	
SIZE This option will display the contents as text in the order of their size.	

0@ THE WINDOW MENU	AFTER FORMATTING
1@drawer NEW DRAWER This menu item allows you to create a new drawer in the selected workbench window.	
1@parent OPEN PARENT Open Parent will open and bring to front the current window's parent window.	
1@close CLOSE This item will close the selected window.	
1@update UPDATE Update will clear the contents of the selected window and read back in the contents icon information.	
1@contents SELECT CONTENTS This item will select the contents of the selected window.	
1@cleanup CLEAN UP This will clean up and display the selected window's icons in a more orderly fashion.	
1@snapshot SNAPSHOT Use one of the two sub-menu items below to save icon information.	
2>window WINDOW This will save the selected window's position and size.	
2@all ALL This will save the selected window's position and size, as well as, the positions of all of the window's icons.	
1@show SHOW This item allows you to either show only icons or all the files in a selected window.	
2@icons ONLY ICONS This option will show only the files with associated icons.	
2@files ALL FILES This option will show all the files. If the window is viewed by icon, all of the files without icons will have temporary icons attached to them.	
1@view VIEW BY There are several ways you can view the contents of the selected window.	
2@icon ICON This option will display the contents as icons.	
2@name NAME This option will display the contents as text in alphabetical order.	
2@date DATE This option will display the contents as text in the order that they were created.	
2@size SIZE This option will display the contents as text in the order of their size.	

Running ASCII2AG.ttx

Once the section tags have been added and the document has been saved, load it into TurboText - if it isn't already - and select "Exec ARexx..." from the "Macros" Menu. A load requester will come up. Select the ASCII2AG.ttx Macro and "_Exec" in the requester.

The Macro will first diagnose the document to find if section tags and NODE names are in order. As previously mentioned, if the macro cannot find the first section tag 0@ it will give the user the option of adding one.

When the macro is finished, the document can now be displayed in AmigaGuide.

```

/*****
(c) Copyright 1992-93 Commodore-Amiga, Inc.
    All rights reserved.

This software is provided as-is and is subject to change;
no warranties are made. All use is at your own risk. No
liability or responsibility is assumed.

$VER: ASCII2AG.ttx 1.00 (07.01.93)
-----
Make an ASCII file into an AmigaGuide database.
File must specify where nodes start by:
<node#><node name> <node title>

Written by Jerry Hartzler
*****/

options results

setinputlock on
getport
workport-result
getfilepath
file-result
getfileinfo
parse var result .. filename
filename=compress(filename," ")

/* Save User prefs and */
/* setup program prefs */

getprefs findbackward
fwr-result
getprefs findignorecase
fwr-result
getprefs findstring
fwr-result
getprefs findwholewords
fwr-result
setprefs findbackward off
setprefs findignorecase on
setprefs findwholewords off

/*****
Check work file for correct node formatting
*****/

err=0
SetStatusBar Temporary 'Error Checking...'
movesof
insertline
moveup
lchar=0

/* Check if there are any nodes */
movesof
find '78'
if rc=0 then do
    message="No nodes found!"
    call error1
end

```

```

else do
    /* Check if 1st node is MAIN (08) */
    getchar
    if result=0 then do
        message="First section tag is not 08"
        call error1
    end
    find '08'
    if rc=0 then do
        /* Check if more MAIN nodes are found */
        message="File must have only one 08"
        call error2
    end
end

do forever /* Check if Nodes are in sequential order */
    find '78'
    if rc=0 then break
    call position
    llin=lin
    if col=1 then do
        getchar
        char=result
        test-datatype(char)
        if test="NUM" then do
            if char<lchar then lchar=char
            if lchar=char & lchar1=char then do
                message="Section numbers not in sequential order"
                call error2
            end
        end
        lchar=char
    end
    moveright 2
    /* Check if two nodes */
    /* Have identical names */
    getchar
    nchar=C2D(result)
    if nchar=32 & nchar=10 then do
        setprefs findignorecase off
        setprefs findwholewords on
        setbookmark 1
        getword
        fword='7' substr(result,2)
        do forever
            find fword
            if rc=0 then break
        end
        call position
        if col=1 then do
            message="Same NODE names: lines' llin-1 '6' llin-1"
            call error2
        end
    end
    setprefs findignorecase on
    setprefs findwholewords off
    movebookmark 1
end
end

/*****
Format work file as AmigaGuide database
*****/
movesof

```



```

text '@DATABASE'      /* Make work file an AG database */
if stnode=1 then do
  insertline
  text '08'
  movesof
end

nodenum=0
endnum=0
lchar=1
do forever
  find 'zg'
  if rc=0 then leave /* exit when zg is not found */
  getcursorpos /* does zg start on 1st column? */
  parse var result . col .
  if col=1 then do
    getchar
    char=result
  end
  if char=0 then do
    node='MAIN'
    end
  end
  else do '@ENDNODE'
    insertline
    end
  delete
  getline 2
  line=result
  call nodename
  moveright 1
  deletetool
  test 'NODE' node name
  movesol
  test-datatype(char)

  if char=0 & test='NUM' then do
    moveup /* Make Link Point */
    setbookmark endnum /* for NODE
    movebookmark char-1
    if strip(name,'b','=')= '' then name=node
    text '@('name,'link',node',' /* if no title */
    insertline /* use name of NODE */
    setbookmark char-1
    movebookmark endnum
    movedown
    endnum=char
    end
  end

moveeof
text '@ENDNODE' /* Insert ENDNODE at end of database */
insertline
call quit

```

```

/******
***** Calls
*****
nodename: /* Parse zg line for NODE info */
  name=''
  if length(line)>0 then do
    if C2D(left(line,1))=32 & node='', then do
      node='subword(line,1,1)' /* Get NODE name */
      line=substr(line,2)
    end
    if length(line)>0 then name=strip(line,'b','=')
    else name='' /* Get NODE title */
  end
  if node='' then do /* If NODE has no name */
    nodenum=nodenum+1 /* make one up
    node='filename',nodenum'
  end
  return

position: /* Find column position */
  getcursorpos
  parse var result lin col .
  return

error1: /* Does user want a MAIN (?) node */
  requestbool TITLE message PROMPT 'Make One? OK=Yes CANCEL=No'
  if result='NO' then call error2
  atnode=1
  return

error2: /* Change work file to the way it was */
  err=1
  setbookmark 1
  movesof
  delete
  movebookmark 1
  beepscreen
  setStatusBar message
  call quit

quit:
  address value workport /* Set back to user Prefa */
  setprefa findbackward fbw
  setprefa findignorecase flc
  setprefa findstring fs
  setprefa findwholewords fw
  setinputlock off
  if err=0 then do
    movesof
    setStatusBar 'Operation Complete'
  end
  exit

```




Overview of V39 Graphics

by Spencer Shanson

The Advanced Amiga Chip set (AA) redefines graphics performance on the Amiga by adding many new display modes, new features and more color to the Amiga platform. This article presents an overview of how the new V39 graphics library has implemented these new features in the system software.

In V39, the graphics library has changed in the following areas:

- ☐ Support for new AA display modes
- ☐ Double buffering
- ☐ Retargettable sprite and screen (ViewPort) functions
- ☐ Palette sharing
- ☐ Bitmap functions
- ☐ Interleaved BitMaps
- ☐ Display mode promotion, coercion and selection
- ☐ RTG compatible RastPort functions.

The original Amiga graphics library exposed many device-dependent details to the application programmer. Because of this, introduction of new graphics devices has been slowed, and application support of new features has been delayed. To reverse this trend, no features have been added to the new graphics system which cannot be kept for the future. Thus, when newer graphics systems are introduced, system software will need fewer changes, and applications will be ready to automatically use new capabilities. Thus, parameters such as number of bits per pixel, resolution, color palette size, etc., are either variable or have been made very large.

Compatibility

The AA chips are register level compatible with the old and ECS chips at boot up time. However, when new AA modes are enabled and displayed, and a game then takes over the screen display without informing the OS of it, some registers may be in incompatible settings.

Two approaches will reduce this:

- 1) Old Programs which boot with their own custom boot block (games) will have AA features disabled unless they are specifically asked for. This should ensure transparent compatibility for all bootable games.
- 2) It will be possible to disable AA features for non-compatible programs. This will be done via the "BootMenu" which is available at system boot time. New options in this menu will allow disabling of AA, disabling of ECS, and switching of PAL and NTSC.

For bootable games that want AA features, use the V39 graphics function SetChipRev(). This lets you upgrade the Amiga to be any chip revision you need, and handles the extra house-keeping that graphics needs to operate the selected chipset (such as updating the graphics database). The only restriction is that it is not possible to downgrade the setting from AA down to ECS or A.

Get/Set Functions

This release of graphics.library will attempt to ease the (future) transition to Retargetable Graphics. New functions are provided to do some operations in a more device-independent manner. This will help when we have to support foreign graphics devices, more than 8 bits per pixel, chunky pixels, and true-color displays.

These new "Get/Set" functions will allow device independent access to fields in the RastPort structure which were only previously manipulable by direct structure access.

VOID SetAPen(rp, pen)	Sets the current APen value
ULONG GetAPen(rp)	Return current APen value
VOID SetBPen(rp, pen)	Sets the current BPen value
ULONG GetBPen(rp)	Return current BPen value
VOID SetDrMd(rp, mode)	Sets the current DrawMode
ULONG GetDrMd(rp)	Return current DrawMode
VOID SetOPen(rp, c)	Was a macro. Now use the SetOutlinePen() function, or the SafeSetOutlinePen() macro.
ULONG GetOPen(rp)	Return current area outline pen
VOID SetWrMsk(rp, m)	Was a macro. Now use the SetWriteMask() function, or the SafeSetWriteMask() macro.
ULONG GetWrMsk(rp)	Description of this function is left as an exercise for the reader.
VOID SetABPenDrMd(rp, apen, bpen, drmd)	Sets the APen, BPen, and DrawMode for a RastPort. Faster than separate calls.

`VOID SetRPAttr(rp, taglist)`

You asked for it so here it is. Lets you set many of the RastPort attributes with one tag-based call. Here are the attributes currently supported:

<code>RPTAG_Font</code>	Font for Text() <code>RPTAG_SoftStyle</code> style for text (see <code>graphics/text.h</code>)
<code>RPTAG_APen</code>	Primary rendering pen
<code>RPTAG_BPen</code>	Secondary rendering pen
<code>RPTAG_DrMd</code>	Drawing mode (see <code>graphics/rastport.h</code>)
<code>RPTAG_OutLinePen</code>	Area Outline pen
<code>RPTAG_WriteMask</code>	Bit Mask for writing.
<code>RPTAG_MaxPen</code>	Maximum pen to render (see <code>SetMaxPen()</code>)

`VOID GetRPAttr(rp, taglist)`

No prizes for guessing what this does. It supports the extra tag `RPTAG_DrawBounds`. This tag is passed with a pointer to a rectangle structure. The returned rectangle structure will contain the bounds of the clipping regions inside the RastPort. This can be used to optimize drawing and refresh.

Color Map Functions

The color palette in AA is different in a lot of ways from the ECS one:

- ☐ It has 24 bits per entry, plus one bit to select transparency.
- ☐ There are 256 entries which is enough for many programs running on the same screen to share the palette.

All colors are now specified as 32 bit left-justified fractions, and are truncated based upon the number of bits that the hardware is capable of displaying.

There are now `...RGB32()` functions to replace the `...RGB4()` functions. These all work in 32-bits per gun, irrespective of the device the colors are intended for. Devices that cannot handle the color resolutions will truncate the colors to the most significant *n* bits. That is why it is important to duplicate the most significant *n* bits throughout the 32-bit resolution. For example, pure white should be treated as:

`R = 0xffffffff, G = 0xffffffff, B = 0xffffffff`

and not

`R = 0xf0000000, G = 0xf0000000, B = 0xf0000000`

The new color palette functions allow for multiple applications to coordinate their access to the palette. This allows applications to, for instance, dynamically remap pictures to match the color palette of the Workbench screen, display animations in windows, etc.

ObtainPen() will attempt to obtain a free palette entry for use by your program, and then set the pen number to the RGB value you specify. A pen can be either shared or exclusively for your own use. If shared, then you should not modify the RGB value of the pen once the pen is obtained, because other applications may use that same pen, and will be relying on its color.

ObtainBestPen() is a tag-based function that will attempt to find the pen in the ColorMap which is closest to the specified RGB value, and then return a shared pen to you. There are (currently) four levels of precision to which the RGB value must match.

PRECISION_EXACT asks for an exact match of the RGB value. The other three are, in descending order of precision, **PRECISION_IMAGE**, **PRECISION_ICON**, and **PRECISION_GUI**. If there is no pen in the ColorMap with the required RGB value to the specified precision, and there are unallocated pens, then a new pen is reserved as shared, and its RGB value set to the value you requested.

Note, there is no way to physically stop you from changing the colors of shared pens, but it's just not done, so don't do it. Pens obtained either with **ObtainPen()** or **ObtainBestPen()** should be released back to the system with **ReleasePen()** when no longer used.

All the palette-sharing functions require a struct **PalExtra** to be attached to the ColorMap. This is done automatically for all Intuition screens, but if you are not using Intuition, then you will need to call **AttachPalExtra()** yourself. This allocates and attaches a **PalExtra** structure to the ColorMap. The **PalExtra** is deallocated by the **FreeColorMap()** function.

FindColor(cm,r,g,b,maxcolor) lets an application find the "closest" color to a given RGB value, independently of palette sharing. In fact, using **SetRGB32CM** and **FindColor**, you can perform color matching which is not associated with a display at all.

Bitmap Functions

These function exist because the new AA chips have alignment restrictions in high bandwidth modes. Changing **InitBitMap()** and **AllocRaster()** to obey these restrictions would have been very incompatible. Bitmaps created by **AllocRaster** with a multiple of 32 or 64 pixels per line will be compatible with high fetch modes (2x or 4x respectively). Incompatible ones will fall back to 1x mode, if 1x mode is capable of displaying the screen.

AllocBitMap() allocates an entire bitmap structure, and the display memory for it.

AllocBitMap() allows you to use more than 8 planes, and also allows you to specify another bitmap pointer, thus telling the system to allocate the bitmap to be "like" another bitmap. A bitmap allocated in such a manner may be able to blit to this bitmap faster. Such a bitmap may be stored in a foreign device's local memory. Do not assume anything about the structure of a bitmap allocated in this manner. The size of a bitmap structure is subject to change in future graphics releases. Thus, you should use **AllocBitMap()/FreeBitMap()** for your raster allocation.

Sprite Functions

Graphics sprite functions (`MoveSprite()`) have been extended to understand large sprites, selectable sprite pixel resolution, and movement of scan-doubled sprites. Sprite positioning is no longer rounded down to lo-res pixel resolution. Applications will no longer have to "know" about the hardware-dependent format of sprite data.

The new sprite functions work with an `ExtSprite` structure, which is obtained with the tag-based `AllocSpriteData()` function. This function allows you to specify a `BitMap` for the image of the sprite, and tags to specify scaling factors to apply to the `BitMap`. The returned `ExtSprite` structure is then passed to `GetExtSprite()`, which is the V39 equivalent of the old `GetSprite()` function. There is no equivalent `FreeExtSprite()` function; `FreeSprite()` does the trick. The `ExtSprite` allocated with `AllocSpriteData()` is freed with `FreeSpriteData()`.

The AA chip set imposes some limitations on sprites:

- ❑ All sprites in a `ViewPort` will be of the same resolution and width. There is no individual sprite resolution/width control. If the sprite you allocate is of a different width than Intuition's pointer, then Intuition is notified, and takes the appropriate steps to maintain the pointer imagery.
- ❑ In the programmable beam modes (namely Multiscan, Euro72, DbINTSC, DbIPAL, and Super72), only sprite 0 is likely to be available; the other sprites are lost to bitplane DMA. `MakeVPort()` has code that detects which sprites have been reserved (with `GetSprite()` or `GetExtSprite()`), and if possible, will drop the display bandwidth. This will make available more DMA to the lower numbered sprites, at the expense of more bitplane DMA access. `MakeVPort()` will not drop the bandwidth if doing so would cause the loss of bitplanes in the display, so this is not always guaranteed to work for you.

All the AA sprite features are available through new tags for `VideoControl()`:

VTAG_SPEVEN_BASE_SET/GET

This sets the base color number of the even numbered sprites. The AA chip set allows odd numbered and even numbered sprites to use the colors of different 16-color banks, as opposed to the previous chip sets where all sprites used colours 16-31. Legal values to set the base are 0-255, but will be rounded down to the nearest multiple of 16.

VTAG_SPODD_BASE_SET/GET

As `VTAG_SPEVEN_BASE_SET/GET`, only for the odd numbered sprites. For example, if the following tag list is passed to `VideoControl()`:

```
struct TagItem[] = {
    {VTAG_SPEVEN_BASE_SET, 32},
    {VTAG_SPODD_BASE_SET, 144},
    {TAG_DONE},
};
```


then the sprites use the following colors:

Sprite	Colors
0	32 (transparent), 33,34,35
2	36 (transparent), 37,38,39
4	40 (transparent), 41,42,43
6	44 (transparent), 45,46,47
1	144 (transparent), 145,146,147
3	148 (transparent), 149,150,151
5	152 (transparent), 153,154,155
7	156 (transparent), 157,158,159

[Attached sprites use the palette settings of the odd numbered sprites.]

Normally, there are only ($2 \ll \text{depth}$) colours in a ColorMap (with the exception that Intuition screens always have a minimum of 32 colours). If you wish to set the sprite's colors to use pens that are outside of this range, then you should use the `VTAG_FULLPALETTE_SET` tag, which specifies that the ColorMap should contain entries for all possible colors (256). Therefore, the colors of these entries can be set with the usual `...RGB32()` functions. Not setting `VTAG_FULLPALETTE_SET` in this case may cause unpredictable colors and enforcer hits.

VTAG_SPRITERESN_SET/GET

This allows you to set all the sprites in the ViewPort to one of 5 resolutions.

SPRITERESN_140NS: All sprites have 140ns pixels.
SPRITERESN_70NS: All sprites have 70ns pixels.
SPRITERESN_35NS: All sprites have 35ns pixels.
SPRITERESN_DEFAULT: All sprites have the Intuition default resolution.
SPRITERESN_ECS: Compatible with ECS resolutions; 140ns, except in 35ns display pixel modes (SuperHires), where the sprite pixels are 70ns.

VTAG_DEFSPRITERESN_SET/GET

For setting the default sprite resolution, as used by `SPRITERESN_DEFAULT`.

VTAG_BORDERSPRITE_SET/GET/CLR:

This sets the bordersprite option in the AA chipset for the ViewPort, which allows sprites to appear in the borders outside of the normal display window (DisplayClip), i.e., the area that is normally color 0. However, this does not apply to the horizontal blanking area.

VTAG_PF1_TO_SPRITEPRI_SET/GET:

All revisions of the Amiga chips have allowed the priorities of sprites to playfield to be set. Usually, the priority is such that the sprites always appear in front of the playfield. There is an entry in the ViewPort structure for altering this priority, but V39 provides this tag in preference to writing to the ViewPort structure.

VTAG_PF2_TO_SPRITEPRI_SET/GET:

As VTAG_PF1_TO_SPRITEPRI_SET/GET, only for playfield 2. On the Amiga, playfield 2 is the default playfield, and playfield 1 is only used in DualPlayfield modes!

Display Mode IDs and the Graphics Database

The graphics database has been extended where necessary for AA information, and these changes are limited to the DisplayInfo structure. The DisplayInfo->PaletteRange is only one WORD long, which is insufficient for the AA 24-bit colour resolution, and so has been superseded by three new entries called RedBits, BlueBits and GreenBits. These are one BYTE each, and show how many bits of precision is available for each colour gun. 255 bits each for red, green and blue should be enough for the foreseeable future (pun intended).

Some new DIPF_IS flags were added to the database, providing more information about each display mode ID.

DIPF_IS_SPRITES_ATT shows the mode supports attached sprites. The 35ns display modes on ECS could not support them.

DIPF_IS_SPRITES_CHNG_RES shows that the mode supports sprites that can change resolution, and so the VTAG_SPRITERESN_SET VideoControl() tag will work on ViewPorts in this mode.

DIPF_IS_SPRITES_BORDER shows that this mode supports border sprites, so the VTAG_BORDERSPRITE_SET VideoControl() tag will work on ViewPorts in this mode.

DIPF_IS_SCANDBL shows that this mode is scandoubled (each display line is repeated once), so that half as many lines take up the same physical area on the monitor. There should be no need for you to look at this bit.

DIPF_IS_SPRITES_CHNG_BASE shows that this mode supports sprite base colour offset changing, so that the VTAG_SPODD/SPEVEN_BASE_SET VideoControl() tag will work on ViewPorts in this mode.

DIPF_IS_SPRITES_CHNG_PRI shows that this mode supports changing the playfield to sprite priority, so that the VTAG_PF2/PF1_TO_SPRITEPRI_SET VideoControl() tag will work on ViewPorts in this mode.

DIPF_IS_DBUFFER shows that this mode will work with the double-buffering ChangeVPBitMap() function. You should check this flag before using the double-buffering functions on the ViewPort.

DIPF_IS_PROGBEAM shows that this mode is a programmed beam-sync mode.

DIPF_IS_FOREIGN shows that this mode is not native to the Amiga chip set. Currently, no mode IDs will have this flag set, but under RTG many 3rd party display devices will. You may want to start checking for this flag now.

With the plethora of new display modes that are available under V38 and V39, it is now becoming harder and harder for both the application writer and the end user to know which display mode ID they need to use. This will be especially true under RTG when the application writer will have no way of knowing all the possible display mode IDs that are available on third party display devices. A function was added for V39 to alleviate this problem, and provide a means by which the system can calculate the best mode ID to use given a number of requirements, called **BestModeID()**. It takes the following tags:

BIDTAG_DIPFMustHave

The **DIPF_** flags that this display mode ID must have set. Default is **NULL**, so there is no preference.

BIDTAG_DIPFMustNotHave

The **DIPF_** flags that this display mode ID must not have set. For example, you may wish to ensure that only native Amiga modes are considered, so use **BIDTAG_DIPFMustNotHave** with **DIPF_IS_FOREIGN**. The default value is defined as **SPECIAL_FLAGS**, which is (**DIPF_IS_DUALPF** | **DIPF_IS_PF2PRI** | **DIPF_IS_HAM** | **DIPF_IS_EXTRAHALFBRITE**) so you may need to **OR** your particular requirements with **SPECIAL_FLAGS**.

BIDTAG_ViewPort

An initializes **ViewPort** for which a mode ID is sought. For example, to find an interlaced version of a **ViewPort**:

```
ID = BestModeID( BIDTAG_ViewPort, ThisViewPort,
                 BIDTAG_MustHave, DIPF_IS_LACE,
                 TAG_END);
```

BIDTAG_NominalWidth

BIDTAG_NominalHeight

Together make up the aspect ratio of the desired mode ID. If specified, will override the Width and Height of the **ViewPort** passed in **BIDTAG_ViewPort**, or the **NominalDimensionInfo** of the ID passed in **BIDTAG_SourceID**. Defaults to 640x200.

BIDTAG_DesiredWidth

BIDTAG_DesiredHeight Nominal width and height of the returned mode ID.

BIDTAG_Depth

Mode ID must support at least this many bitplanes. Defaults to **vp->RasInfo->BitMap->Depth** of the **ViewPort** passed in **BIDTAG_ViewPort**, or 1.

BIDTAG_MonitorID

The returned mode ID must belong to this monitor family.

BIDTAG_SourceID

BestModeID() will use characteristics of this mode ID, and override some of the characteristics with the other values in the taglist.

BIDTAG_RedBits

BIDTAG_BlueBits

BIDTAG_GreenBits

The mode ID must support at least these many bits for each color gun. Defaults to 4 bits each, so A2024 modes will not be considered.

As an example of its use, here is a portion of the code for the **V39 CoerceMode()** function (which is used by Intuition when coercing screens):

```
ULONG CoerceMode(struct ViewPort *vp, ULONG MonitorID, ULONG Flags)
{
    ...
    /* Coerce the ViewPort vp to the best fit ModeID in the monitor
     * MonitorID.
     */
    must = NULL;
    mustnot = SPECIAL_FLAGS;
    tag[t].ti_Tag = BIDTAG_ViewPort;
    tag[t++].ti_Data = vp;
    tag[t].ti_Tag = BIDTAG_MonitorID;
    tag[t++].ti_Data = MonitorID;

    if (GetDisplayInfoData(NULL, (APTR)&dinfo,
                          sizeof(struct DisplayInfo), DTAG_DISP, ID))
    {
        must = (dinfo.PropertyFlags & SPECIAL_FLAGS);
        mustnot = (SPECIAL_FLAGS & ~must);
        if ((Flags & AVOID_FLICKER) && !(dinfo.PropertyFlags & DIPF_IS_LACE))
        {
            /* we don't want lace if AVOID_FLICKER is set, and this
             * ViewPort is not naturally laced.
             */
            mustnot |= DIPF_IS_LACE;
        }
        tag[t].ti_Tag = BIDTAG_RedBits;
        tag[t++].ti_Data = dinfo.RedBits;
        tag[t].ti_Tag = BIDTAG_GreenBits;
        tag[t++].ti_Data = dinfo.GreenBits;
        tag[t].ti_Tag = BIDTAG_BlueBits;
        tag[t++].ti_Data = dinfo.BlueBits;
    }

    tag[t].ti_Tag = BIDTAG_DIPFMustNotHave;
    tag[t++].ti_Data = mustnot;
    tag[t].ti_Tag = BIDTAG_DIPFMustHave;
    tag[t++].ti_Data = must;
    tag[t].ti_Tag = TAG_DONE;

    return(BestModeIDA(tag));
}
```

As another example, consider an IFF display program with a HAM image, to be displayed in the same monitor type as the Workbench ViewPort.

```
ID = BestModeID(BIDTAG_NominalWidth, IFFImage->Width,
                BIDTAG_NominalHeight, IFFImage->Height,
                BIDTAG_Depth, IFFImage->Depth,
                BIDTAG_DIPFMustHave, DIPF_IS_AM,
                BIDTAG_MonitorID, (GetVPMODEID(WbVP) & MONITOR_ID_MASK),
                TAG_END);
```

The definitions of the display mode ID keys have been moved from <graphics/displayinfo.h> to a new <graphics/modeid.h> file.

The include file <graphics/modeid.h> specifically says that the individual bits of the mode IDs should not be checked for any meaning, but that the database should be read to glean information about the mode ID. In order to maintain compatibility with old software, and make the application writer's job somewhat easier, I am willing to guarantee that any mode ID with the bit 0x00000800 set is a HAM mode, and any mode ID with the bit 0x00000080 set is an ExtraHalfBrite mode. These are the only bits that are guaranteed to mean anything in the mode ID itself. These bits correspond of course to the HIRES and EXTRA_HALFBRITE definitions in <graphics/view.h>.

Display Mode Promotion

Promotion is a software solution for the lack of hardware deinterlacing circuitry on the AA machines. With the promotion feature enabled in Icontrol, the default monitor (NTSC on NTSC machines, PAL on PAL machines), becomes DblNTSC on NTSC machine and DblPAL on PAL machines. There are a number of advantages and gotchas with this approach.

One advantage is that the graphics database is always truthful. Any enquiries about a default monitor mode ID will yield information relevant to whatever the default monitor happens to be at the time. This should not be a problem for any code written for V37 onwards, as the default monitor has always been either NTSC or PAL; now, there are just more possibilities. Another advantage is that V39-aware software that requires NTSC or PAL modes (e.g., video titling software), can now get such modes using specific NTSC or PAL mode IDs.

Here is a list of "gotchas" (that is, things to watch out for if you want display mode promotion to work correctly).

- 1) The default monitor is dynamic, and can change at any time. Therefore, do not cache any information about the default mode IDs, but read them from the database as you need them.
- 2) There is no equivalent in the Dbl... monitors to the SuperHires modes. The database LVOs sniff out SuperHires mode IDs for database enquiries, and map SuperHires mode IDs to the equivalent Dbl... Hires mode IDs if promotion is enabled.

- 3) Programs that rely on copper timings for UserCopperlists, such as SHAM displays, may no longer work when promoted, because each line is shorter in time than the NTSC/PAL line. Therefore, there are less coppercycles per line.
- 4) Promoted ViewPorts have less sprites available than non-promoted ViewPorts.
- 5) For compatibility, we do not promote 1.3 style custom ViewPorts and Views (we check for the presence of a ViewExtra).

Miscellaneous

Interleaved screens have been added. These use a different layout of the graphics data in order to speed rendering operations and eliminate the annoying "color-flash" problem which is the hallmark of planar (as opposed to "chunky") display architectures. Set the `BMF_INTERLEAVED` flag when calling `AllocBitMap()`.

Double buffering functions have been added. These allow applications to display very smooth, synchronized animation in fully an efficient "Intuition-friendly" manner. Call `AllocDBufInfo()` to allocate and initialize the `DBufInfo` structure, which is then passed to `ChangeVPBitMap()`. The double-buffering functions return up to two different types of messages. The first (`dbi_SafeMessage`) tells your program when it is safe to write to the old `BitMap`. The second (`dbi_DispMessage`) is sent when it is safe to call `ChangeVPBitMap()` again and be certain the new bitmap has been seen for at least 1 field. The autodocs for `AllocDBufInfo()` has example code showing how to safely double buffer with `ChangeVPBitMap()`.

The `DBufInfo` structure should be freed with the `FreeDBufInfo()` function.

Due to the extra colours that need to be loaded in deeper AA screens, the gap between screens can now be greater than the three lines that Intuition traditionally kept. In fact, the number of lines between screens is variable, depending on the screen type and depth. A new function `CalcIVG()` (`CalcInterViewPortGap`) has been added to calculate the number of lines required by the copper to execute all the copper instructions before the display window is opened. Note however that `CalcIVG()` returns the true number of lines (rounded up to the next whole line) needed in ViewPort resolution units, but Intuition still maintains a gap of at least three lines between Screens. Therefore, when calling `CalcIVG()` to position screens in an Intuition environment, use the result of `MAX((laced ? 6 : 3), CalcIVG(v, vp))`.

There is one other caveat with respect to `CalcIVG()`. This function works by counting the number of copper instructions in the `ViewPort->DspIns` list, which is set up by `MakeVPort()`. If an Intuition screen is opened behind, then `MakeVPort()` is not called on that screen's ViewPort until it first becomes visible, so calling `CalcIVG()` with that screen's ViewPort will yield a result of 0.

Some operations have been sped up: RectFill() has been rewritten, WritePixel() uses the CPU (3x speedup) ScrollVPort is 10 times faster, and other optimizations.

Bugs Anomalies

There is a big bug in the V37-V39 graphics hash-table code, which is used to associate a ViewExtra with a View; namely, only 8 Views can have ViewExtras attached to them. This has been fixed in the latest SetPatch and Kickstart.

ScrollVPort() and ChangeVPBitMap() have problems with 8-bit HAM mode. This has been SetPatch'ed.

Many other outstanding bugs have been fixed for V39.

Plans for 3.01

Some features that are planned for Release 3.01:

- ☐ An LVO for the games writers to handle color fades with user copperlists, and to allow independent scrolling of parts of the ViewPort for parallax scrolling games.
- ☐ Frame rate control for ChangeVPBitMap().
- ☐ Add tags to BestModeID() for overscan considerations.
- ☐ Still faster ScrollVPort (next beta).



Intuition 3.0

by Peter Cherna

Graphics and AA-Chipset Issues

Support for New Modes

Intuition now has direct support for the AA display modes, through extensions to old mechanisms and through new tags. This includes the ability to select higher resolutions and to set colors in better than 4 bits-per-gun. The graphics database and the ASL screen-mode requester are the definitive places to get information about what modes and depths are available or desired. You can specify higher depths using SA_Depth, and new display modes with SA_DisplayID. SA_Colors32 supplants SA_Colors for specifying colors with a higher precision than 4 bits-per-gun. For full details, see the section on new screen features.

Mode Promotion

The AA chipset provides some flicker-free display modes that are roughly equivalent to NTSC or PAL when output through a display-enhancer or flicker-fixing product. While the AA chipset provides these modes without the significant extra expense of a display-enhancer, some software help is required. Conversely, the display-enhancer lives on the video output, and is completely transparent to software.

To use promotion, the user needs to have a multiscan monitor. He needs to install the DbI NTSC or DbI PAL monitor into his devs:monitors drawer, and ensure that the "Mode Promotion" option in IControl Prefs is on (starting with 3.01, this setting will be on by default, eliminating this last step). When this is done, the graphics database entries for the "default" monitor will map to the most appropriate modes from the DbI NTSC (or DbI PAL) monitor, in place of the NTSC (or PAL) monitor.

Interlaced screens that are promoted are displayed using double-height non-interlaced modes. For example, 640x400 NTSC interlaced appears as 640x400 double-NTSC non-interlaced. Non-interlaced 15 kHz screens are promoted using a chipset feature called "scan-doubling," in which each scan-line is output twice, because the 200 or 256 lines of the original screen only fill half the screen when in 31 kHz modes.

Most applications which use the default monitor (either by explicitly using mode-IDs such as

HIRES_KEY or by using V34-style 16-bit mode descriptions) and which go through Intuition and have relatively ordinary display requirements will be successfully and transparently promoted, and their screen will be displayed flicker-free. Applications which refer to modes of explicit monitors (e.g., SA_DisplayID of NTSC_MONITOR_ID|HIRES_KEY) are never promoted. The preferred method for an application is to use the ASL screen-mode requester, which normally presents all explicit modes (including the modes of DbINTSC and DbIPAL) and omits the modes of the "default" monitor.

Promotion and Compatibility

De-interlacing in hardware is expensive but transparent, since the Amiga chipset's operation is unchanged by the presence of such hardware. However, promotion can change the behavior of the system in a manner which may be incompatible with certain applications. These are:

- ☐ The overscan limits of DbINTSC (DbIPAL) are a little less than the overscan limits of NTSC (PAL). (For 3.01, the DbINTSC (DbIPAL) limits have been extended and are now comparable to NTSC (PAL)).
- ☐ It may be harder to center a DbINTSC (DbIPAL) screen on certain multiscan monitors than it was to center a de-interlaced NTSC (PAL) screen. (3.01 provides additional centering flexibility which basically solves this problem).
- ☐ An interlaced screen is promoted to a non-interlaced screen, which has obvious implications on custom copper-lists.
- ☐ The higher resolutions/depths of the AA chipset require higher alignment restrictions on bitplanes. Fortunately, most applications either let Intuition allocate their screen's BitMap or else they have a custom BitMap whose width is a multiple of 64 pixels (the highest alignment currently required by AA). However, if the custom BitMap is an unusual width, it may not be sufficiently aligned for the hardware. Such a screen can come up skewed when promoted.

"1x" modes require 16-pixel (word boundary) alignment of each scan-line. "2x" modes require 32-pixel (longword boundary) alignment, while "4x" modes require 64-pixel (double-longword boundary) alignment. Here is a short reference:

- ☐ 140 ns pixels (lores in 15 kHz modes, extra-lores in 31 kHz modes)
1-8 planes require 1X

- ❑ 70 ns pixels (hires in 15 kHz modes, lores in 31 kHz modes)
 - 1-4 planes require 1X
 - 5-8 planes require 2X
- ❑ 35 ns pixels (super-hires in 15 kHz modes, hires in 31 kHz modes)
 - 1-2 planes require 1X
 - 2-4 planes require 2X
 - 5-8 planes require 4X

As the graphics.library AllocBitMap() function takes care of allocating suitably-aligned BitMaps for you, you do not need to worry about alignment when using modern system calls.

- ❑ The AA hardware does not allow dual-playfield non-interlaced screens to be scan-doubled, so they will appear half as tall as their non-promoted counterparts.
- ❑ Like earlier chipsets, the AA chipset still supports eight sprites. In much the same way as ECS and original chipsets lose sprites when overscan is increased, many of the new modes have insufficient spare cycles to fetch data for these sprites. A promoted screen may have fewer sprites left than the corresponding 15 kHz mode, meaning that some sprites other than the pointer sprite may vanish.
- ❑ There is no 31 kHz mode having 1280 pixels per line. That would require 17.5 ns pixel speeds, which is twice what the AA chipset is capable of. Therefore, SuperHires screens are promoted to 640 pixel-per-line screens, which generally can scroll.
- ❑ Custom ViewPorts are not promoted, but a graphics-database aware application could open a ViewPort in DbINTSC or DbIPAL.

Pointer Sprite Features

New Pointer Features

Intuition's handling of the pointer sprite has undergone significant rework for V39, partly to add support for AA sprites (35/70/140 ns sprite pixels, 16/32/64 pixels per sprite, scan-doubling of sprites), and partly to make some general improvement to Intuition.

The Intuition pointer now supports the various new sprite modes of the AA chipset. This includes 16, 32, and 64-bit wide sprites, as well as the sprite-pixel resolution control.

Intuition automatically positions the pointer sprite on screen pixel resolution boundaries, even if the pointer is in a lower resolution, with the exception that graphics.library only allows positioning the pointer on every second line of an interlaced screen.

For applications, there is a new boopsi class called "pointerclass," which is used to create Intuition pointer objects. The new <intuition/pointerclass.hli> include file contains definitions for the attributes that pointerclass supports, including:

- ☐ POINTERA_BitMap - BitMap to use for sprite imagery
- ☐ POINTERA_XOffset, POINTERA_YOffset - sprite hot-spot
- ☐ POINTERA_WordWidth - intended width in words of this pointer
- ☐ POINTERA_XResolution, POINTERA_YResolution - intended resolution of this pointer

The resolution can be any of the hardware resolutions (ECS-compatible, 140 ns, 70 ns, or 35 ns), but Intuition also adds software-managed choices for

- ☐ sprite resolution to match screen pixel resolution
- ☐ sprite resolution to be "always lores" (~320 pixels per line)
- ☐ sprite resolution to be "always hires" (~640 pixels per line)

See <intuition/pointerclass.hli> for full details.

There is a new pointer-control function called SetWindowPointerA(), which takes a window and a taglist. The tag values are as follows:

- ☐ WA_Pointer (APTR) - used to specify an application custom pointer, ti_Data points to an instance of "pointerclass" you typically obtain using NewObject(). If NULL, you are requesting the Preferences default pointer.
- ☐ WA_BusyPointer (BOOL) - if ti_Data is TRUE, this tag requests the Preferences default busy pointer.
- ☐ WA_PointerDelay (BOOL) - if ti_Data is TRUE, this tag requests that the change of pointer imagery be deferred for a short duration. This is very useful for an application which is about to be busy for an unknown but possibly very short duration. Such an application should request both the Preferences default busy pointer and the pointer-delay feature. If the application clears the pointer or sets another pointer before the delay expires, the pending pointer change is cancelled. This reduces

short flashes of the busy pointer caused by the application having brief intervals of busy-ness.

The same three tags are now recognized by `OpenWindowTagList()`, so you can now arrange for a window to open with a custom pointer (or standard busy pointer) already in place, and never see a brief flash of the default pointer.

The user can now specify 32 pixel wide pointers in hires or lores using Pointer Preferences. Pointer Preferences supports a user-defined default pointer image as well as a user-defined busy pointer image.

Intuition blanks the pointer around size or imagery changes, to reduce ugly flashing that might otherwise result.

On an upbeat note, the notorious off-by-one error in sprite hot-spot position was eliminated from all new Intuition and Graphics calls. When using the new mechanisms, always specify the correct hot-spot offset. (The truth is, we tried to leave the error in, but couldn't agree whether the error should be one lores pixel or one sprite-resolution pixel.)

Sprite Compatibility

The old `SetPointer()` and `ClearPointer()` functions still work, giving compatible Intuition pointers. Likewise, the pointer imagery in `devs:system-configuration` will be used until a `pointer.prefs` file is received. However, due to growing complexity in the pointer subsystem, calling `SetPointer()` or `ClearPointer()` from within an input handler or inside `Begin/EndRefresh()` runs a risk of deadlocking. This has been kludged around, however:

This is why we warn people to stick to simple rendering functions only!
While inside `LockLayerInfo()`, `LockLayer()`, `BeginRefresh()`, `BeginUpdate()`, etc., and to not call Intuition or other high-level system functions inside of `LockIBase()`. As well, great care should be taken inside an input handler (it's generally best for the handler to signal a high-priority task which actually does the work). Don't be the application that dies under the next release when an inappropriate function that happened to work now deadlocks because its handling has become more sophisticated.

See the Autodocs for these functions for more information on what other system calls are OK to use with these functions.

Intuition and Graphics are involved in a scheme to maximize compatibility with older sprite-using applications. The AA chipset supports variable sprite resolution and width, but the setting affects all sprites. If an application requests a specific sprite width (through the old `graphics.library/GetSprite()` call or the new `graphics.library` calls (`GetExtSpriteA()` and `ChangeExtSpriteA()`), and Intuition's sprite is not compatible with that request, then `graphics.library` will blank Intuition's sprite and notify Intuition. Intuition will rebound by generating the most suitable pointer sprite which is compatible.

Using an attached sprite for the Intuition pointer was quasi-supported under 2.0. It no longer works.

The pointer information returned by `GetPrefs()` is no longer kept up-to-date, since the pointer data can exceed the storage space available in struct Preferences. (The ROM default pointer will be returned in all cases). (Like V37, V39 ignores the pointer data in calls to `SetPointer()` after the first one, for reasons such as this).

Pen-Sharing Support

Under V39, `graphics.library` has functions that let multiple applications share the pens in a palette (`ObtainBestPen()`, etc.). Palette sharing allows an application to gain exclusive access to some palette entries, which that application may then use or change as it sees fit. As well, an application may access a pen as shareable, meaning that other applications in need of a similar color may also be granted that pen value. Because these pens are shared among several clients, applications may not alter their color.

Intuition now uses and supports this pen-sharing scheme. For all types of screen, the sprite pens and pens found in the `DrawInfo->dri_Pens` are obtained as shareable. By default, all other pens are allocated as exclusive on behalf of the screen opener (this provides compatibility when visitor windows aware of the pen-sharing functions open on unaware public screens). Exclusive pens are for the screen owner's use only, and may be changed at the owner's will.

Screens that are aware of pen-sharing issues should set the `{SA_SharePens,TRUE}` tag, which instructs Intuition to leave all other pens unallocated. The Workbench screen is so marked, but pens 0 to 3 and ~0 to ~3 are also made shareable, for compatibility. Screens with `SA_SharePens` set to `TRUE` will have the new `PENSHARED` bit set in the `screen->Flags` field.

The application may then allocate pens as needed. A paint package, for example, would allocate all colors it uses as exclusive. Other applications might allocate several colors as

shared or exclusive. Since Intuition opens all public screens in private state, the application has a chance to allocate its colors before making the screen available to visitors (see `SetPubScreenModes()`).

Preferences now listens to only 8 colors for the BitMap, which Intuition will set as the first four and the last four colors of the Workbench or any "Workbench-like" screen (those having the `SA_FullPalette` or `SA_LikeWorkbench` attributes). When the `SA_Pens` pen-array is evaluated, pens are masked to the number of available colors. As well, special definitions of pen-number (`PEN_C3`, `PEN_C2`, `PEN_C1`, and `PEN_C0`) mean the complementary pen numbers of pens 0 to 3, regardless of depth.

The way the DrawInfo pens are determined is Intuition picks a default pen-array. Then, any pens you supply with `SA_Pens` override the defaults, up until the ~0 in your array. If the screen is monochrome or old-look, the default will be the standard two-color pens. If the screen is two or more planes deep, the default will be the standard four-color pens. If any explicit pens are specified, the default colors for new-look menus and the titlebar match the V37 colors. If the `SA_Pens` tag points at ~0, the new-look menu colors will be used.

If the screen has the `SA_LikeWorkbench` property, the default will be the user's preferred pen-array, now changeable through Preferences. There is a preferred pen-array for four colors, and one for eight or more colors.

Miscellaneous Graphics-Level Changes

Intuition places a blanking gap between sliding screens. The time spent in this gap is used to load the hardware color registers, display-mode registers, and bitplane-pointers in a clean way. Under ECS and prior, three non-interlaced lines were sufficient, and this amount was hard-coded. Under AA, this amount may increase, so logic was added to `graphics.library` (`CalcIVG()`) to determine this amount. Intuition now bases its inter-screen gap on the result of this call.

Under 2.0, when a screen was coerced, Intuition determined what display mode to use based on tables built into Intuition. This was too limiting, so `graphics.library` now has a `CoerceMode()` function to fulfill this responsibility. Intuition now uses this function. (Interested persons should see `graphics.library/BestModeIDA()`, which has wider application than `CoerceMode()`).

Enhancements to Screens

Attached Screens

It is becoming increasingly popular for an application to have multiple screens open simultaneously. A typical use is an application that has a full-sized screen (perhaps in HAM mode) as a canvas, and a short screen as a control panel or palette. Since it is desirable that these screens slide and depth-arrange together, Intuition now provides the ability to attach screens together. `OpenScreenTagList()` now supports the `SA_Parent` tag to attach a new child to an existing parent. The `SA_FrontChild` and `SA_BackChild` tags can be used to attach an existing child in front of or behind a new parent. When opening a parent screen you can specify multiple `SA_xxxChild` tags, in order to attach multiple children. Draggable child screens can be dragged independently of each other and their parent, except that they can never go above their parent. Pulling down the parent below its natural top causes the child screens to move as well.

Attached screens always remain adjacent to each other in the screen depth-ordering. It is not possible to interpose some other screen between screens of the same family. User depth-arrangement (via Amiga-M/N or the screen depth-gadget), as well as old programmatic depth-arrangement (`ScreenToFront()` and `ScreenToBack()`), depth-arrange a screen's family as a single unit, moving them to the front or to the back of the list of screens without altering the ordering of screens within the family. The new `ScreenDepth()` function has an `SDEPTH_INFAMILY` option which allows the programmer to depth-arrange screens within their family.

There are times when it would be useful for a parent and child screen to masquerade as a single screen. This can allow independent setting of screen mode, depth, resolution, etc. Set the new `SA_Draggable` tag to `FALSE` to get a child screen which is non-draggable with respect to its parent. Trying to drag a child screen (through `MoveScreen()`, its drag-bar, or mouse-screen-drag) is equivalent to dragging the parent. The new `ScreenPosition()` function has an `SPOS_FORCEDRAG` option which allows the application to independently move such a child screen.

To complement attached screens, a feature called "menu lending" has been implemented. This allows menu button presses in one window to activate the menus of a different window, and have the menus appear in the screen of that other window. The idea is to allow unification of the menu strips of attached screens. The `LendMenus()` function is used for this.

If `OpenScreen()` is unable to attach screens (due to illegal hierarchies, etc.), the screen will

fail to open, with a secondary error of `OSERR_ATTACHFAIL`. A parent screen may not itself have a parent, nor may a child screen have a child. Also, a child screen may not be the child of more than one screen. One parent may legally have several child screens.

Aside from the `SA_Parent`, `SA_FrontChild`, or `SA_BackChild` tags, and the drag and depth arrangement behavior described, attached screens are just like other screens, that is to say they live on the regular screen list (`IntuitionBase->FirstScreen-> ...`), and child screens don't inherit any properties from their parent. In particular this means you can run the same code under Release 2 and all you will lose are the depth arrangement and dragging relationships. That would yield the best situation under those versions, and no conditional code is required on your part.

Various combinations of `DClips` and over-sized scrolling screens are supported by attached screens, and they work in much the manner you would expect. Note that there are problems when non-draggable child screens are attached to parent screens and their `DClips` or dimensions are not equivalent. These problems are fixed in 3.01.

The `attachdemo.c` example on the DevCon disks shows usage of attached screens.

Double-Buffering Support for Screens

Intuition now supports double (or multiple) buffering inside an Intuition screen, with full support for menus, and support for certain kinds of gadget.

The `AllocScreenBuffer()` call allows you to create other `BitMap` buffers for your screen. the `SB_SCREEN_BITMAP` flag is used to get a buffer handle for the `BitMap` currently in use in the screen. Subsequent buffers are allocated with the `SB_COPY_BITMAP` flag instead, which instructs Intuition to copy the current imagery (e.g., the screen title-bar and any of your rendering) into the alternate `BitMap`. Normally you let Intuition allocate these alternate `BitMaps`, but if your screen is `CUSTOM_BITMAP`, you should allocate the alternate `BitMaps` yourself.

To swap buffers, call the `ChangeScreenBuffer()` function, which attempts to install the new buffer. `ChangeScreenBuffer()` will fail if Intuition is temporarily unable to make the change (say while gadgets or menus are active). Intuition builds these functions on top of the new `graphics.library` `ChangeVPBitMap()` function, so the signalling information that graphics provides is also available to the Intuition user. To clean up, call `FreeScreenBuffer()` for each screen buffer. It is not necessary to restore the original buffer before freeing things. Consult the autodocs for full details.

When the user accesses the screen's menus, buffer-swapping will stop. The `ChangeScreenBuffer()` call will return failure during this time. When the user finishes his menu selection, buffer-swapping will be possible again. Only a small subset of gadgets are supportable in double-buffered screens. These gadgets are those whose imagery returns to the initial state when you release them (e.g., action buttons or the screen's depth gadget). To use other kinds of gadgets (such as sliders or string gadgets) you need to put them on a separate screen, which can be an attached screen.

Windows with borders are not supportable on double-buffered screens. Double-buffered screens are expected to consist nearly entirely of custom rendering.

An example program illustrating double-buffering under Intuition, with menu-lending and an attached screen to hold two slider gadgets is provided.

Miscellaneous Screen Features

The new `{SA_Interleaved,TRUE}` tag allows applications to request that their custom or public screen have an "interleaved" BitMap. Interleaved BitMaps are built out of a single allocation, instead of one per bitplane. Further, the data is laid out as follows:

```
bitplane 0, scan-line 0
bitplane 1, scan-line 0
...
bitplane n, scan-line 0
bitplane 0, scan-line 1
...
```

(Contrast this to regular BitMaps, where each bitplane is contiguous.) The primary advantage of interleaved BitMaps is that blitting between them is cleaner, because color artifacting is eliminated. As well, multiple small blits are replaced by fewer large blits, saving on blitter setup time and improving blitter/processor overlap.

The primary disadvantage of interleaved BitMaps is that `BitMap->BytesPerRow` no longer means "how many bytes are in one row of one bitplane." This field continues to mean "how many bytes must be added to the current address to arrive at the same pixel one row down." See the notes for the compatibility talk for further details. (Screen grabbers and screen printers seem to be the primary victims of this change.)

For compatibility, the Workbench screen is non-interleaved if it opens before `IPrefs` has run. If opened or reset after `IPrefs` has run, the Workbench screen will be interleaved.

The new `SA_LikeWorkbench` tag gives you a screen having the same attributes as the Workbench screen, including depth, colors, pen-array, screen mode, etc. Individual attributes can be overridden through the use of tags. (`SA_LikeWorkbench` itself overrides things specified in the `NewScreen` structure). Attention should be paid to hidden assumptions when doing this. For example, setting the depth to two makes assumptions about the pen values in the `DrawInfo` pens. Note that this tag requests that Intuition *attempt* to open the screen to match the Workbench. There are fallbacks in case that fails, so it is not correct to make enquiries about the Workbench screen then make strong assumptions about what you're going to get.

`OpenScreen()` now allocates an appropriate-sized `ColorMap` for new modes. The `SA_ColorMapEntries` tag can override this, to let the application increase the `ColorMap` size if needed.

The `SA_Draggable` and `SA_Exclusive` tags are designed to help implement "game screens" that coexist with Intuition. `{SA_Draggable,FALSE}` allows the caller to make a screen non-draggable. `{SA_Exclusive,TRUE}` allows the caller to make a screen exclusive, meaning it will never share the display with another screen. Dragging down a screen that's in front of an exclusive screen won't reveal the exclusive screen. Although exclusive screens can autoscroll, but they can't be pulled down below their natural top (since nothing should be visible behind). Starting with 3.01, you can attach one or more exclusive screens with the `SA_Parent` and `SA_xxxChild` tags already described. Such screens form an exclusive family, and only coexist on the display with each other, not with other screens (exclusive or not).

Intuition and `graphics.library` now support over-sized scrollable `A2024/Moniterm` screens. While such screens can autoscroll and be dragged around, they can't be pulled down, since other screens would not be visible behind them (similar to the `SA_Exclusive` property, but dictated by the `graphics` database `MonitorInfo.Compatibility` field).

The new `ScreenPosition()` function extends the functionality of `MoveScreen()` to also include absolute screen positioning and optional movement of `{SA_Draggable,FALSE}` screens. The hope is that only the screen's opener (and not any commodity-type programs) would forcibly move a non-draggable screen. As well, `ScreenPosition()` allows you to specify a rectangle in screen coordinates which you wish to be made visible. Over-sized screens will be scrolled such that the rectangle you supply will be on the visible part of the display.

The new `ScreenDepth()` function unifies `ScreenToFront()` and `ScreenToBack()` while adding a flag allowing optional in-family depth-arrangement of attached screens. `ScreenToFront()`, `ScreenToBack()`, Amiga-M/N, screen depth-gadget action, and `EasyRequest` screen-popping all go through the `ScreenDepth()` LVO. Due to a bug, `WBenchToFront()` and `WBenchToBack()` do not yet do so, but this is expected to be fixed for 3.01.

OpenScreen() and the routines which open the Workbench now call **OpenScreenTagList()** through the LVO, which will allow some useful **SetFunction()**ing.

OpenScreen() now supports the new **SA_BackFill** tag, to install a **LayerInfo** backfill hook for the screen.

Intuition now ensures that the Workbench screen is at least 640x200. This is needed in order to safely allow lores (and other odd resolution) Workbench screens.

The original **Screen** structure has an embedded instance of a **BitMap** structure, which is unfortunate. When Intuition allocates a screen's **BitMap** (i.e., a non-CUSTOMBITMAP screen), it now uses **AllocBitMap()**, and copies the struct **BitMap** into **&screen->BitMap**. This is the direction to head for RTG, and it is needed now for double-buffering. Intuition internally now references the real **BitMap** (obtainable as **screen->RastPort.BitMap**) instead of **&screen->BitMap**. It is recommended that applications do the same.

The new **SA_Colors32** tag can be used to provide 32-bit color information at **OpenScreen()** time. **ti_Data** points to a longword-array that Intuition will pass to **LoadRGB32()**. See the autodoc for **LoadRGB32()** for details on how to format the data.

The new **SA_VideoControl** tag allows an application to provide a taglist which Intuition will pass to **VideoControl()** after opening the screen. This can be useful to turn on border-sprites, for example.

The requested screen depth is now validated. Making a request for a screen too deep causes failure with a secondary error of **OSERR_TOODEEP**.

Enhancements to Windows

The window depth-gadget now determines whether a click should send the window to front or to back based on whether the window is obscured or not, rather than on whether it's the top layer.

The new **ScrollWindowRaster()** function implements **ScrollRasterBF()** at the Intuition level. You will receive an **IDCMP_REFRESHWINDOW** event if there is damage to your window. Damage will appear if obscured parts of a simple refresh window are scrolled into view.

Note: this area is not cleared!

When you supply the "alternate-size" rectangle for zooming using the **WA_Zoom** tag, you can now specify (-1,-1) for the upper-left corner. This instructs Intuition to perform size-only

zooming. Wherever the window is placed, zooming will toggle size but not affect position (unless moving the window would be required to keep it on-screen). Using (-1,-1) under V37 is safe, and equivalent to using (0,0).

The Window->WindowPort is now allocated at OpenWindow() time, even if IDCMPFlags of zero are requested. This simplifies ModifyIDCMP() failure handling, because failure can no longer happen when you set the window->UserPort to your shared port, then call ModifyIDCMP() to turn on messaging. ModifyIDCMP() can now only fail if you ask it to create the UserPort.

The new OpenWindow() tag WA_NotifyDepth allows a window to request IDCMP_CHANGEWINDOW messages when that window is depth-arranged. These messages arrive with an IntuiMessage->Code value of CWCODE_DEPTH to distinguish them from V37-style IDCMP_CHANGEWINDOW messages (sent in response to window movement or resizing), which have a Code value of CW_MOVE_SIZE.

When inactive, window borders are filled with BACKGROUNDPEN instead of pen zero. There are presumably a few more places when pen zero is being used incorrectly, but they are hard to track down (ever try to search 1.5 megabytes of source code for all references to "0"?).

The WA_HelpGroup and WA_HelpGroupWindow tags allow the programmer to identify multiple windows of the same application, for purposes of gadget help processing. See the section on gadget help for details.

Enhancements to Gadgets and Imagery

Extended Gadget Structure

V39 introduces the ExtGadget structure as a compatible substitute that can be used wherever old Gadget structures can be found. An arbitrary gadget can be identified as an ExtGadget if the GFLG_EXTENDED bit in its Flags field is set. Never attempt to read any of the extended fields of a gadget if this flag is not set. Starting with V39, all instances of gadgetclass or its subclasses are ExtGadgets. The extended fields include a new longword worth of flags, and a bounding box.

Gadget Bounding Box and GM_LAYOUT

Until now, the only area that was defined for gadgets was the select box area. However, the imagery of a gadget often extends outside its select box. For example, the border of a string

gadget is often outside, as is its label. ExtGadgets have four new fields that describe the "bounding box." The bounding box can be used to allow relative size or position gadgets to work even if they have imagery outside the select box. The BoundsLeftEdge, BoundsTopEdge, BoundsWidth, and BoundsHeight fields of an ExtGadget are assumed valid if the GMORE_BOUNDS flag in the ExtGadget->MoreFlags field is set. Where Intuition wants to use a bounding box, but the gadget is not extended or does not have GMORE_BOUNDS set, the gadget select box will be used instead, which matches the V37 behavior.

The routine that manages erasing and redrawing GRELxxx gadgets during window resizing now bases its work on the gadget bounding box (if one is specified), instead of the gadget select box. This means that you can finally have GREL gadgets which have imagery (e.g., a gadget label) extending outside of the select box, and Intuition will correctly move or resize such a gadget.

As well, there is a new boopsi method for gadgets called GM_LAYOUT. If your gadget has any of the GREL properties, it will receive a GM_LAYOUT message when the gadget is first added (or the window containing the gadget is first opened), as well as whenever the window size changes. At GM_LAYOUT time, the gadget can change its gadget select box and/or bounding box. It can re-allocate or change its image dimensions if it likes. If it is a group-gadget, it can move its members around. To round this all out, there is a new "special relativity" property, GFLG_RELSPECIAL or GA_RelSpecial. Unlike the older GREL properties, this property doesn't affect the interpretation of the gadget box. It does allow your gadget to receive GM_LAYOUT messages, hence have arbitrary layout power.

The DevCon disks include a sample program called relspecial.c that implements a boopsi gadget whose size is kept at half the current size of the window, and which is centered in that window. The example makes use of GFLG_RELSPECIAL and GM_LAYOUT.

Gadget Help

Intuition now supports "gadget help". If a window enables this feature and the user passes the mouse over the bounding box of a gadget which has the GMORE_GADGETHELP property, then an IDCMP_GADGETHELP event will be sent. There is a corresponding boopsi GM_HELPTEST method which boopsi gadgets can use to refine their help-sensitivity areas or to delegate help-testing to member gadgets. Boopsi gadgets can also return values for the IntuiMessage Code field of the IDCMP_GADGETHELP message.

The gadget help feature may be turned on or off through the HelpControl() function.

The gadget help checking is optimized for performance. If the mouse is moving quickly, Intuition skips the check for gadget help. If Intuition discovers that the mouse is still over same gadget as the last one that sent gadget help, no new IntuiMessage is sent unless the gadget wants to report a different IntuiMessage->Code value.

When the mouse is over a GMORE_GADGETHELP gadget, the IDCMP_GADGETHELP message has an IntuiMessage->IAddress which points to the gadget. When the mouse is over the window but not over any help-aware gadget, the IAddress points to the window itself. When the mouse is not over the window, the IntuiMessage IAddress will be zero. Intuition will look "through" gadgets that do not have the GMORE_GADGETHELP property to see if some other gadget lies underneath.

Ordinarily, gadget help only applies to the active window. However, a multi-window application can mark all its windows as being in a group (using the WA_HelpGroup or WA_HelpGroupWindow tags), which makes Intuition test gadget help in all windows of the group when any one of them is the active one. There is a new utility.library function called GetUniqueID() which must be used to provide an ID for WA_HelpGroup. If you have only one window, there is no need to pass WA_HelpGroup. HelpControl() sets the state of gadget help for all windows of a group, and Intuition ensures that all windows of the same group are consistently set.

```
helpgroup = GetUniqueID();

for ( each window )
{
    win[x] = OpenWindowTags(...,
                           WA_HelpGroup, helpgroup, TAG_DONE);
}
```

Inactive windows whose WA_HelpGroup matches the active window's are also subject to gadget help testing. IDCMP_GADGETHELP messages are sent to the window the mouse is over. The IDCMP_GADGETHELP message with an IAddress of zero means the mouse is not over the active window or any other window of the same group. This particular message is always sent to the active window (which is not necessarily the window in your group that last got a message).

All system gadgets (e.g., close, drag, size, depth) have GMORE_GADGETHELP set, so GadgetHelp-aware applications can (almost must) provide help on them too. You can check the gadget->GadgetType & 0xF0 field for GTYP_CLOSE, etc. Later include files define GTYP_SYSTYPEMASK to be this value (0xF0).

The gadgethelp.c example on the DevCon disks illustrates the correct handling of IDCMP_GADGETHELP messages.

Gadget Support for ScrollRaster() Damage Handling

Intuition now notices and repairs damage when boopsi gadgets use ScrollRaster(). (Such damage occurs when the gadget is in a simple-refresh window and part of the scrolled area is obscured). Such gadgets must set GMORE_SCROLLRASTER in order to benefit from this magic repair feature. Note that ScrollWindowRaster() is for applications. Boopsi gadgets must not use ScrollWindowRaster(), but rather use ScrollRaster() or ScrollRasterBF().

Miscellaneous Gadget Features

There is an important new boopsi function in Intuition called DoGadgetMethodA(), which invokes the specified method, but includes a valid GadgetInfo structure if possible. SetGadgetAttrsA() has been re-implemented to go through DoGadgetMethodA(). Two SetGadgetAttrsA() bugs were fixed in the process. First, if a requester is off-window, it has no layer. SetGadgetAttrsA() of a gadget in such a requester wasn't being sent to the gadget, but now OM_SET is now sent with a GadgetInfo of NULL. Second, there was no locking around the call. DoGadgetMethodA() is preferable to a direct boopsi invocation (namely DoMethod()) because it offers both a valid GadgetInfo structure and arbitration around other gadget activity.

The default string edit hook now ignores Return or Enter keystrokes that have the repeat-qualifier set.

PROPNEWLOOK proportional gadgets in the borders of an active window have their knobs rendered in FILLPEN instead of SHINEPEN. When you click on the knob, they become SHINEPEN as before. This looks a lot better, is more consistent with inactive windows, and finally restores that all-important feedback to these prop gadgets.

Imagery Features

Under V37 and prior, Intuition handled ghosting a gadget by blasting a pattern of dots over its select area. This did not allow boopsi images to manage their own disabled rendering. For V39, Intuition has defined a new read-only attribute for image classes, called IA_SupportsDisable. If an image class returns TRUE in response to an OM_GET request of this attribute, it is asking to take responsibility for performing ghosting based on image state. When a gadget is first added to a window, Intuition will check its image to determine if it has the IA_SupportsDisable attribute. When such a gadget is disabled, Intuition skips its own disabled rendering, and draws the image using DrawImageState() using the IDS_DISABLED or the new IDS_SELECTEDDISABLED state.

The ROM boopsi image class for rendering frames, "frameiclass", now supports the standard frame types used by GadTools and recommended by the *Amiga User Interface Style Guide*, including the standard GadTools-style bevelled box, the GadTools string-gadget ridge, and the AppWindow icon drop-box specified by the Style Guide.

The imagery for the standard system arrows has been improved.

The GadTools checkbox and GadTools radio-button images are now marked as scalable, to allow GadTools to support scaled checkboxes and radio-buttons.

The basic gadget types (classic prop, bool, string) now support GFLG_LABELIMAGE.

Enhancements to Menus

New-Look Title Bar and Menus

On aware screens, the title-bar has a nicer appearance.

For V39, Intuition defines three additional pens in the DrawInfo, which are used to control the rendering of the screen title bar and menus. These pens are:

- ☐ BARDETAILPEN - pen used for details like text in the title bar and text or graphics in menus.
- ☐ BARBLOCKPEN - pen used to fill the solid areas of the title bar and menus.
- ☐ BARTRIMPEN - pen to use for the trim-line under the screen title bar.

It is intended that BARDETAILPEN and BARTRIMPEN should be black or dark, and that BARBLOCKPEN should be white or light-colored.

The handling of defaults is a bit involved because of compatibility issues. Applications that specify no SA_Pens array or ones who specify an SA_Pens array with at least one explicit pen provided get compatible defaults. Applications whose SA_Pens array consists solely of {~0} will get the new colors. Finally, screens which specify SA_LikeWorkbench will get the user's preferred pen-array, which allows the user to control the menu and title bar colors.

Under V37 and earlier, the menus themselves are drawn using the window's DetailPen and BlockPen, while the colors of the MenuItems are determined by the imagery (IntuiTexts or Images) chosen. For compatibility, old applications will have their menus rendered in V37-compatible colors. Applications will want to take advantage of new-look menus when

present, but they must request them through the `{WA_NewLookMenus,TRUE}` tag. This instructs Intuition to use `BARDETAILPEN` and `BARBLOCKPEN` for rendering the elements of your menus, instead of relying on the window's `DetailPen` and `BlockPen`. Note that the application (or any menu-building library) should ensure that the colors of struct `IntuiText` or struct `Images` used by `MenuItems` use matching pens. You can instruct `GadTools` to use the new colors by passing `{GTMN_NewLookMenus,TRUE}` to `LayoutMenusA()`.

Intuition also ensures that the Amiga-key and checkmark symbols are colored to match the menu and are scaled to match the screen's font. If you are using a font other than the screen's font for the menus, you must create custom Amiga-key and checkmark symbols using "sysiclass". This image class recognizes a new tag, `SYSIA_ReferenceFont`, which you can use to set the size of a checkmark or Amiga-key symbol appropriately for your chosen font. You can then use the (V36) `WA_CheckMark` tag or the new `WA_AmigaKey` tag to override the imagery Intuition will use in the menus.

Per screen, the default Amiga-key and checkmark images used will be appropriately colored and scaled to the screen's font. (You can find pointers to their imagery in the `DrawInfo` structure for that screen).

HIGHCOMP menu items in new-look menus complement in such a way that pixels colored in `BARDETAILPEN` highlight into `BARBLOCKPEN`, and vice-versa.

The flag which indicates whether a window is using new-look menus is publicly readable (`WFLG_NEWLOOKMENUS`).

Other Enhancements

Drawing Tablet Support

As an input device, a drawing tablet poses special problems for Intuition and for the tablet driver writer. Some of the problems include getting auxiliary information such as pressure through the `IntuiMessage` channel and providing suitable scaling. As well, the absolute nature of tablet devices poses some interesting problems for features like autoscroll. V37 added a simple tablet input event, but this was not enough. There now is a new subclass of `IECLASS_NEWPOINTERPOS` called `IESUBCLASS_NEWTABLET`. This subclass solves tablet handling quite nicely. The tablet driver fills out a few tablet-oriented properties (like the current value and range in X and Y), and then submits the `InputEvent` to `input.device`. Later, Intuition establishes the active screen and the rectangle to which the tablet should scale itself, then calls back the driver through a hook. This allows the tablet driver to handle screen

resolution changes, oversized scrolling screens, pulled down screens, and attached screens. The tablet driver can scale according to some tablet preferences settings it manages (for example, preserve aspect, center, best fit horizontal, best fit vertical). Intuition supplies reasonable default scaling for simple tablet drivers that leave the hook NULL. See struct `INewTablet` in `<devices/inpuevent.h>`.

Windows that request the new `WA_TabletMessages` property receive extended `IntuiMessages`, which include a pointer to a `TabletData` structure. If this `IntuiMessage` originated from a tablet event, the `TabletData` pointer will be non-NULL, and the structure will have some information such as sub-pixel position. In addition, there is a pointer to a tag-list, and there are definitions for standard tags such as pressure, tilt, additional buttons, and a Z-coordinate. The definition and important comments on the `TabletData` structure can be found in `<intuition/intuition.h>`.

As an example, pressure is passed in with the `TABLETA_Pressure` tag. the pressure reading of the stylus. The `ti_Data` member is the pressure, which should be normalized to fill a signed long integer. Typical devices would not generate negative pressure, but the possibility is not precluded. The Preferences program shipped with a tablet driver for a pressure-sensitive device might offer two pressure sensitivity settings. The "contact threshold" would be the pressure below which no contact should be reported by the driver. This is the "zero point" for reported pressure. The "click threshold" would be the pressure at which a button transition should be reported (by setting the `InputEvent ie_Code` to indicate a downpress of the select button). The tablet would send position/pressure events even when the pressure was below the click threshold (but above the contact threshold, of course).

When a tablet driver sends a new tablet event and the active window is tablet-aware, `IDCMP_MOUSEMOVE` events are sent to that window even if the pixel-level mouse-position is unchanged. This is to allow applications to hear changes in sub-pixel position or other parameters such as pressure. This means that tablet drivers must be careful to only send events when something actually changed.

If an input event is a tablet event, boopsi gadgets can get a pointer to the `TabletData` structure in the `gpInput` structure they receive through the `GM_GOACTIVE` and `GM_HANDLEINPUT` messages.

Miscellaneous Features

The new `TimedDisplayAlert()` function allows for alerts that time-out without user-intervention. Exec uses this new function to aid unattended operation of the Amiga, particularly in kiosk and video applications.

In struct Preferences, the unused WorkName[] field is now split into PrtDevName[], DefaultSerUnit, and DefaultPrtUnit, for multi-serial preferences and more flexible printer-preferences.

OpenScreenTagList(), OpenScreen(), OpenWindowTagList(), OpenWindow(), and the internal BorderPatrol() routine now go through stack-swapping. This protects applications from increased stack usage in these calls.

The Preferences LaceWB field (and whether the pretend screen mode from GetScreenData() is lace or not) now solely depends on the height of the text overscan rectangle of the true mode of the Workbench. This helps older applications opening on modes such as double-NTSC 640x400.

Intuition now handles MakeVPort() failure. Intuition will blank any failed ViewPort, and attempt to remake it at each opportunity. MakeScreen(), RethinkDisplay(), and RemakeDisplay() now have return codes that reflect MakeVPort() failure.

Rendering Optimizations

Several important rendering optimizations make Intuition appear snappier and cleaner. Here is a partial list:

- ☐ When a WFLG_ACTIVATE window is opened, Intuition now activates it synchronously. The big benefit is that the window's border is no longer drawn inactive then activate.
- ☐ EasyRequests and AutoRequests used to consist of a window with a requester inside, which meant two layers. This consumed memory and slowed down requester and other window operations. Now, the gadgets and imagery are brought up directly in the window, saving a layer.
- ☐ The window sizing/dragging rubber-band box is now much faster.
- ☐ The bar-layer of each screen no longer does any backfill processing, since Intuition fully re-renders the screen bar anyway.
- ☐ Intuition now avoids spurious border and gadget refresh and sending a spurious IDCMP_REFRESHWINDOW event. Formerly, if an application had not cleared its window's damage when another damage-causing operation (e.g., menus, window sizing/movement) occurred on the same screen, another round of refresh was performed. Intuition now can tell if the window's layer had been damaged since the last IDCMP_REFRESHWINDOW message went out.

- ☐ **Menus** are brought on-screen somewhat faster than before, and are removed very much faster than before. (3.01 and up only)
- ☐ A lot of work has been done to reduce window border and gadget flashing during window resize operations. (3.01 and up only)

Bug Fixes

NextPubScreen() could write a zero-byte one byte past the end of the buffer the caller supplies. It no longer does this.

Clicking in the no-window area of a screen now makes that screen the active screen (for purposes of autoscrolling).

Fixed a long-standing bug where **REQCLEAR** messages weren't being sent when a requester having no layer is taken down while other requesters are still up in the window.

Setting a negative minimum width or height with **WindowLimits()** no longer allows you to crash the computer by turning a window inside out.

SetWindowTitles() now properly erases remnants of the previous title, even when odd extenders happen.

A ghosted string gadget no longer causes patterning in string gadgets that precede it in the list and have a non-zero container background-pen.

NewModifyProp() of a disabled prop gadget no longer clears away part of the ghosting.

SetGadgetAttrs() to a proportional gadget no longer can cause the window's installed clip-region to be lost.

The mouse-pointer no longer blanks in the first gap when three interlaced screens up.

Several causes of sprite-pointer jumping have been fixed.

The ROM default sprite pointer now is the 2.0/3.0 one, instead of the 1.3 one.

DClips of coerced **ViewPorts** are finally scaled as correctly as possible.

If you used a boopsi string gadget as an integer gadget, with Intuition supplying the buffer,

and you specified a `STRINGA_MaxChars` of `> 15`, you would get a mismatched `FreeMem()` when the gadget is disposed. This is now fixed.

There is a bug in 3.0 (not in 2.0x and fixed in 3.01) where the autoscroll boundary was inadvertently switched to be the DClip of the active screen, where it used to be the "hull" of the DClips of all the screens. If there are two screens in the system with different DClips, the mouse can be way outside the DClip of the smaller screen. If that screen is active, it will `AutoScroll` at a ridiculous rate. For example, if the mouse is seventeen pixels below its DClip, moving it down one pixel causes the screen to autoscroll by eighteen, instead of one. This is now fixed.

Starting with 3.01, Intuition now updates its internal time values based on (nearly) *any* `InputEvent` it receives, instead of just `IECLASS_TIMER` ones. The problem was that outgoing `IntuiMessages` get their time from this internal time, which meant that `IntuiMessage` time was the time-stamp of the most recent timer tick, instead of the time-stamp of the event that actually triggered this `IntuiMessage`. This problem completely precludes correlating an `IntuiMessage` with the `InputEvent` that caused it, which is important for tablet people, for example. A `SetPatch` for earlier ROM versions is being considered.

In 3.00, there is a bug where the part of a window obscuring the title-bar area of a `SCREENQUIET` screen wasn't erased when the window was closed or moved away. Effectively, Intuition was relying on a layers side-effect that was optimized out for V39. Intuition fixes this for 3.01.

Starting with 3.01, when Intuition splits a single `InputEvent` into button and movement components, the button event is now sent first. This fixes some inconsistencies with extended input information like pressure, as well as odd behavior of the qualifiers, in particular `IEQUALIFIER_MIDBUTTON`.





DataTypes

by David N. Junod

Introduction

DataTypes provides an object oriented approach for determining data types and handling those data types.

There are many advantages to using DataTypes:

- ❑ An application can detect what data type a file is and handle it accordingly. An example of this would be a BBS that examines incoming files and labels them by file type so that the appropriate integrity checks can be applied or the appropriate contents viewer invoked for that archive type. Or a directory utility that invokes that appropriate editor for the data type of the file that the user selects.
- ❑ An developer can easily embedded a DataType viewer within their application without worrying about writing a lot of code. For example, if an application needs to display a picture, or word-wrapped text in a proportional font, it can do it easily using the functions of DataTypes.
- ❑ An developer can easily add clipboard support to their application using the functions of DataTypes. Each root DataType class implements clipboard support in the native Amiga format for that type of data. All the application needs to do is submit its data to a DataType object and invoke the copy method.
- ❑ Applications can handle multiple formats of a data type transparently. For example a Paint package can handle ILBM, GIF, Windows BMP or any other type of picture data as long as there is a DataTypes class to handle it.
- ❑ A data viewer can transparently handle any type of data. An example of this is the MultiView utility that comes with 3.0 or the ClipView utility on the examples disk.
- ❑ It is easy to write a sub-class for most of the data types, because all the class implementor needs to do is convert the data to the internal Amiga format for that data type. For example, the picture class handles the color remapping of a 256 color picture to the screen depth of the destination Amiga.

- ❑ The **DataType** objects have a consistent interface. There is no difference between displaying a picture, text, or animation.
- ❑ A **DataType** object can be queried to see what methods and commands they support.

Determining Data Type

One of the main features of the **DataTypes** system is its ability to determine the data type of a block of data. This data block can reside in a file or the clipboard.

The following functions are used to determine the **DataType** of a data block:

ObtainDataTypeA()

Obtain the **DataType** descriptor for a data block.

ReleaseDataType()

Release the **DataType** descriptor for a data block.

The data type detection functions use the **DataType** structure.

```
struct DataType {
    struct Node      dtn_Node1;
    struct Node      dtn_Node2;
    struct DataTypeHeader *dtn_Header;
    struct List      dtn_ToolList;
    STRPTR           dtn_FunctionName;
    struct TagItem   *dtn_AttrList;
    ULONG            dtn_Length;
};
```

The **DataType** structure is read-only. The only pertinent field is the **dtn_Header** field, which points to a **DataTypeHeader** structure.

```
struct DataTypeHeader {
    STRPTR    dth_Name;
    STRPTR    dth_BaseName;
    STRPTR    dth_Pattern;
    WORD      *dth_Mask;
    ULONG     dth_GroupID;
    ULONG     dth_ID;
    WORD      dth_MaskLen;
    WORD      dth_Pad;
    UWORD     dth_Flags;
    WORD      dth_Priority;
};
```

The **DataTypeHeader** structure fields are as follows:

dth_Name

Descriptive name of the data type. For example, the description for an ILBM data type could possibly be "Amiga BitMap Picture".

dth_BaseName

This is the base name for the data type and is used to obtain the class that handles this data type.

dth_GroupID

This indicates the main type data that the object contains. Following are the possible values:

GID_SYSTEM	Fonts, Executables, Libraries, Devices, etc...
GID_TEXT	Formatted or unformatted text.
GID_DOCUMENT	Formatted text with embedded DataTypes (such as pictures).
GID_SOUND	Audio samples.
GID_INSTRUMENT	Audio samples used for playing music.
GID_MUSIC	Musical scores.
GID_PICTURE	Graphic picture or brush.
GID_ANIMATION	Moving picture or cartoon.
GID_MOVIE	Moving picture or cartoon with sound.

dth_ID

This is an individual identifier for the DataType. For IFF files it is the same as the FORM type, for example ILBM for an Amiga BitMap picture. For non-IFF files, it is the first four characters of **dth_Name**.

dth_Flags

The flags field contains, among other information, the coarse type of data. The type can be obtained by ANDing **DTF_TYPE_MASK** with this field.

DTF_IFF	Interchange File Format
DTF_BINARY	Non-readable characters
DTF_ASCII	Readable characters
DTF_MISC	Disks and drawers

dth_Pattern dth_Mask dth_MaskLen dth_Priority

These fields are used by the detection code in **datatypes.library** for determining the data type. See the "Defining a DataType Descriptor" section below for more information.

Following is a code fragment that shows how to determine the data type of a file. This fragment uses functions from `datatypes.library`, `dos.library`, and `iffparse.library`.

```
STRPTR name = "somefilename";
BPTR lock;

struct DataTypeHeader *dth;
struct DataType *dtn;
UBYTE idesc[5];
STRPTR tdesc;
STRPTR gdesc;
UWORD ttype;

/* Obtain a lock on the file that we want information on */
if (lock = Lock (name, ACCESS_READ))
{
    /* Get a pointer to the appropriate DataType structure */
    if (dtn = ObtainDataTypeA (DTST_FILE, (APTR) lock, NULL))
    {
        /* Get a pointer to the DataTypeHeader structure */
        dth = dtn->dtn_Header;

        /* Get the coarse type */
        ttype = dth->dth_Flags & DTF_TYPE_MASK;

        /* Get a pointer to the text strings */
        tdesc = GetDTString (ttype + DTMSG_TYPE_OFFSET);
        gdesc = GetDTString (dth->dth_GroupID);

        /* Convert the ID to a string. */
        IDtoStr (dth->dth_ID, idesc);

        /* Display the information */
        printf ("    Description: %s", dth->dth_Name);
        printf ("    Base Name: %s", dth->dth_BaseName);
        printf ("    Type: %d - %s", ttype, tdesc);
        printf ("    Group: %s", gdesc);
        printf ("    ID: %s", idesc);

        /* Release the DataType structure now that we are done with it */
        ReleaseDataType (dtn);
    }

    /* Release the DOS lock on the file */
    UnLock (lock);
}
```

Embeddding DataTypes

Since `DataType` objects are a sub-class of the `Intuition Gadget` class, `DataType` objects can be attached to an `Intuition` window in a similar way that gadgets can be added to a window.

DataTypes use a parallel set of functions because it requires additional information that the Intuition functions weren't able to provide.

Currently the handling of the data is limited to reading, writing, printing, viewing (audio or visual), and clipboard access. The MultiView utility is an example of an application that embeds DataType objects.

The following functions are used to access DataType objects.

NewDTObjectA()

Obtain a handle on a DataType object.

DisposeDTObject()

Release the handle on a DataType object.

SetDTAttrsA()

Set the attributes of a DataType object.

GetDTAttrsA()

Get attributes of a DataType object

AddDTObject()

Add a DataType object to a window.

RefreshDTObjectA()

Refresh the rendering of a DataType object.

RemoveDTObject()

Remove a DataType object from a window.

GetDTMethods()

Get a list of the methods that a DataTypes object supports. Write, Copy, and Select are examples of methods that an object may support.

GetDTTriggerMethods()

Get a list of the trigger methods that a DataType object supports. An action like Play, Pause, and Resume are examples of trigger methods that an object may support.

DoDTMethod()

Invoke a DataTypes method.

PrintDTObject()

Asynchronously print a DataType object.

GetDTString()

Get the localized text string for a DataTypes text id. Useful for obtaining localized error messages.

Creating a DataType Object

The DataTypes function `NewDTObjectA()` must be used to create a new DataType object.

```
dto = (Object *) NewDTObjectA (APTR name, struct TagItem *attrs)
```

The pointer that `NewDTObjectA()` returns is a pointer to a BOOPSI object. Like other BOOPSI objects, DataType objects are "black boxes" and are not to be peeked and poked without using the provided interface.

To create a DataType object, `NewDTObjectA()` needs to know where to obtain the data used to create the object. By default the name is treated as a filename.

The attrs tag list is a list of tag/value pairs, each of which contains an initial value for an attribute. There are a number of attributes defined in `<datatypes/datatypesclass.h>` that can be used at creation time.

DTA_SourceType

Specifies the type of the source data. The default is `DTST_FILE`. The types are:

DTST_RAM

Source data is in RAM.

DTST_FILE

Source data is a file. Name is the name of the file.

DTST_CLIPBOARD

Source data is in the clipboard. Name is the unit number. For example, `(APTR)0`, for clipboard unit zero.

DTST_HOTLINK

Reserved for future use.

DTA_Handle

Can be used instead of the name field. If the source type is `DTST_FILE` then handle must be a valid BPTR file handle. If the source type is `DTST_CLIPBOARD` then handle must be a valid IFFHandle.

DTA_DataType

Can be used to specify the class for handling the data. Data must be a pointer to a valid DataType. This should only be used when attempting to create a new object that doesn't have source data, or could be handled by multiple classes (for example, could be used to force an object to be handled by the AmigaGuide class).

DTA_GroupID

If this tag is present, then the data must be of the specified type, or the object creation will fail with `ERROR_OBJECT_WRONG_TYPE`. This can be used by a Sound editor to ensure that only sounds can be loaded, for example.

DTA_TextAttr

Specify the font to use for any text rendered by this object.

There are additional attributes that can be specified at creation time, but are dependant on the data type of the object being created. See the header files `<datatypes/#!/class.h>` for more attributes.

The following attributes, which are defined in `<intuition/gadgetclass.h>`, are also valid.

GA_Left

Specify the left edge of the gadget.

GA_RelRight

Specify the left edge of the gadget being relative to the right edge of the containing window.

GA_Top

Specify the top edge of the gadget.

GA_RelBottom

Specify the top edge of the gadget being relative to the bottom edge of the containing window.

GA_Width

Specify the width of the gadget.

GA_RelWidth

Specify the width of the gadget being relative to the width of the containing window.

GA_Height

Specify the height of the gadget.

GA_RelHeight

Specify the height of the gadget being relative to the height of the containing window.

GA_ID

Specify a ID associated with the gadget.

GA_UserData

Attach application data to the gadget.

GA_Immediate

Indicate that the application should be notified of gadgetdown events for this gadget.

GA_RelVerify

Indicate that the application should be notified of gadgetup events for this gadget.

GA_Previous

For adding the gadget to a list of gadgets.

GA_DrawInfo

A pointer to a struct DrawInfo.

In order for the application to receive information from the DataType object, it must set up a target for the notification attributes that the object sends out.

ICA_TARGET

Specify a target for the notification attributes that the DataType object sends out.

ICA_MAP

Specify an attribute mapping for the notification attributes that the DataType object sends out.

The usual method to obtain notification is to set up an ICA_TARGET of ICTARGET_IDCMP so that the application will receive the attributes via the IDCMP_IDCMPUPDATE Intuition message class. But it is also possible to set up another BOOPSI object as the receiver.

If NewDTObjectA() is successful, it returns a pointer to a DataType object. Otherwise it returns NULL and the reason for failure can be obtained using IoErr(). See the Autodocs for datatypes.library for more information.

Disposing of a DataType Object

When the application is done with the DataType object it has to dispose of the object. To dispose of a DataType object, you must use the DataTypes function DisposeDTObject():

```
VOID DisposeDTObject (Object *dto)
```

where dto is a pointer to the DataType object to be disposed. This will abort any PLAY trigger method, such as playing sounds or animations, but will wait for a print or save to complete.

Obtaining Environment Information for a DataType

In order to embed a DataType object in a window, it is necessary to ask the object what its minimum environment is. For example, since the remap code doesn't handle remapping HAM pictures, they must be shown on a HAM screen, and therefore can't be added to a window that is on a non-HAM screen.

```
ULONG modeid = INVALID_ID;
LONG nomwidth, nomheight;
BOOL useScreen = FALSE;
struct dtFrameBox dtf;
struct FrameInfo fri;

/* Get the attributes that we are interested in */
GetDTAttrs (dto,
            PDTA_ModeID,           &modeid, /* Get the mode ID */
            /* Get the desired size */
            DTA_NominalHoriz,      &nomwidth,
            DTA_NominalVert,       &nomheight,
            TAG_DONE);

/* Clear the structures */
memset (&dtf, 0, sizeof (struct dtFrameBox));
memset (&fri, 0, sizeof (struct FrameInfo));

/* Fill in the message */
dtf.MethodID = DTM_FRAMEBOX;
dtf.dtf_FrameInfo = &fri;
dtf.dtf_ContentsInfo = &fri;
dtf.dtf_SizeFrameInfo = sizeof (struct FrameInfo);

/* Perform the frame method */
if (DoDTMethodA (dto, NULL, NULL, (Msg) &dtf)){
    /* Check to see if the object requires a HAM screen */
    if (fri.fri_PropertyFlags & DIPF_IS_HAM) {
        printf ("HAM");
        useScreen = TRUE;
    }
    /* Check to see if the object requires an ExtraHalfBrite screen */
    else if (fri.fri_PropertyFlags & DIPF_IS_EXTRAHALFBRITE){
        printf ("ExtraHalfBrite");
        useScreen = TRUE;
    }
    /* A safety check to see if a screen is required */
    else if ((fri.fri_PropertyFlags == 0) && (modeid & 0x800)
              && (modeid != INVALID_ID)) {
        printf ("ModeID=0x%08lx", modeid);
        useScreen = TRUE;
    }
}
else
{
    /* No special environment required, can be attached to any screen mode */
}
```

Adding a DataType Object to a Window

A DataType object must be added to a window using the AddDTObject() function of DataTypes.

```
LONG AddDTObject (struct Window *w, struct Requester *r, Object *dto,
                  LONG pos)
```

This function will add a DataTypes object to the existing gadget list for the specified window. The recommended value for pos is -1 which will cause the DataType object to be added to the end of the list.

DataType objects should not be added using the WA_Gadgets attribute to OpenWindowTagList() or by using the AddGList() function. There is special information that DataTypes requires that will not be obtained if any method other than AddDTObject() is used.

When the DataType object is added to the window, the layout method for the object will be invoked. It is possible that the layout will take a while to perform, in that case the object will spawn a process to handle the layout asynchronously. In order to refresh the object's visual information, it is necessary to obtain IDCMP_IDCMPUPDATE messages from the object and refresh the object when a DTA_Sync attribute is received.

The following code fragment illustrates adding a DataType object to a window.

```
Object *dto;

struct IntuiMessage *imsg;
struct Window *win;
ULONG sigr;

struct TagItem *tstate, *tags;
ULONG tidata;
ULONG errnum;

BOOL going = TRUE;

/* Set the pertinent attributes of the DataType object */
SetDTAttrs (dto, NULL, NULL,

            /* Set the dimensions of the object */
            GA_Left,    win->BorderLeft,
            GA_Top,     win->BorderTop,
            GA_Width,   win->Width - win->BorderLeft - win->BorderRight,
            GA_Height,  win->Height - win->BorderTop - win->BorderBottom,
```

```

        /* Make sure we receive IDCMP_IDCMPUPDATE messages from the object */
        ICA_TARGET, ICTARGET_IDCMP,
        TAG_DONE);

/* Add the object to the window */
AddDTObject (win, NULL, dto, -1);

/* Refresh the DataType object */
RefreshDTObjects (dto, win, NULL, NULL);

/* Keep going until we're told to stop */
while (going)
{
    /* Wait for an event */
    sigr = Wait ((1L << win->UserPort->mp_SigBit) | SIGBREAKF_CTRL_C);

    /* Did we get a break signal */
    if (sigr & SIGBREAKF_CTRL_C)
        going = FALSE;

    /* Pull Intuition messages */
    while (imsg = (struct IntuiMessage *) GetMsg (win->UserPort))
    {
        /* Handle each message */
        switch (imsg->Class)
        {
            case IDCMP_IDCMPUPDATE:
                /* Get a pointer to the attribute list */
                tstate = tags = (struct TagItem *) imsg->IAddress;

                /* Step through the attribute list */
                while (tag = NextTagItem (&tstate))
                {
                    tidata = tag->ti_Data;
                    switch (tag->ti_Tag)
                    {
                        /* Change in busy state */
                        case DTA_Busy:
                            if (tidata)
                                SetWindowPointer (win, WA_BusyPointer, TRUE,
                                                    TAG_DONE);
                            else
                                SetWindowPointer (win, WA_Pointer, NULL,
                                                    TAG_DONE);
                            break;

                        /* Error message */
                        case DTA_ErrorLevel:
                            if (tidata)
                            {
                                errnum = GetTagData (DTA_ErrorNumber, NULL,
                                                        tags);
                                PrintErrorMsg (errnum,
                                                (STRPTR)options[OPT_NAME]);
                            }
                    }
                }
            }
        }
    }
}

```

```

        }
        break;

        /* Time to refresh */
        case DTA_Sync:
            /* Refresh the DataType object */
            RefreshDTObjects (dto, win, NULL, NULL);
            break;
    }
}
break;
}

/* Done with the message, so reply to it */
ReplyMsg ((struct Message *) imsg);
}
}

```

Removing a DataType Object from a Window

A DataType object must be removed from the window using the `RemoveDTObject()` function.

```
LONG RemovedTObject (struct Window *w, Object *dto)
```

This function removes the DataType object from the window's gadget list.

This is the only way that a DataType object should be removed from the window list. Using `RemoveGList()` is not supported, nor is removing the object manually.

Setting an Existing DataType Object's Attributes

An object's attributes are not necessarily static. An application can ask an object to set certain attributes using the `SetDTAttrs()` function.

```
ULONG SetDTAttrsA (Object *dto, struct Window *w,
                  struct Requester *r, struct TagItem *attrs)
```

The return value is DataType object specific, but generally a non-zero value means that the object needs to be visually refreshed.

The following fragment illustrates how to set the current top values for a DataType object, using the VarArgs version of `SetDTAttrsA()`.

```
SetDTAttrs (dto, window, NULL,
            DTA_TopVert, 0,
            DTA_TopHoriz, 0,
            TAG_DONE);
```

This will cause the DataType object to update its vertical and horizontal top values. If the object has been added to a window, then the display will be updated accordingly.

Note that it is not okay to call SetGadgetAttrs() or SetAttrs() on a DataType object.

Getting a DataType Object's Attributes

The DataTypes function GetDTAttrsA() is used to obtain the values for a list of attributes from a DataType object.

ULONG GetDTAttrsA (Object *dto, struct TagItem *attrs)

Where dto is a pointer to a DataType object returned by NewDTObjectA().

And attrs is a TAG_DONE terminated array of attributes. Where the data element of each pair contains the address of the storage variable for that attribute.

This function will return a number that indicates that number of attributes that it was able to obtain. For example if four attributes asked for and GetDTAttrs returns a four, then all the attributes were obtained.

The following code fragment illustrates how to get the current top values for a DataTypes object using the VarArgs form of GetDTAttrsA().

```
LONG topv, toph;

if (GetDTAttrs (dto, DTA_TopVert, &topv, DTA_TopHoriz, &toph, TAG_DONE) == 2)
{
    printf ("Top: vertical=%ld, horizontal=%ld", topv, toph);
}
else
{
    ("couldn't obtain the top values");
}
```

Writing A Sub-Class

For each of the DataType categories, there is a class that handles the data. Handlers for explicit data are sub-classes under the appropriate class. For example, a class that handles

ILBM pictures would be a sub-class of the Picture class.

In order to fully understand the class concept used by DataTypes it helps to have a basic understanding of BOOPSI.

A sub-class must provide an OM_NEW method that converts the source data into the data format required by the super-class. The sub-class must optionally have a OM_DISPOSE method that discards any of the data constructed in the OM_NEW method. All other methods must be passed to the super-class.

Following is a listing of a example class dispatcher for a sub-class of the picture class:

```
ULONG ASM Dispatch (REG (a0) Class *cl, REG (a2) Object *o, REG (a1) Msg msg)
{
    struct ClassBase *cb = (struct ClassBase *) cl->cl_UserData;
    ULONG retval;

    switch (msg->MethodID)
    {
        case OM_NEW:
            if (retval = DoSuperMethodA (cl, o, msg))
            {
                /* Convert the source data to required data format */
                if (!GetObjectData (cb, cl, (Object *)retval,
                                    ((struct opSet *) msg)->ops_AttrList))
                {
                    /* Force disposal of the object */
                    CoerceMethod (cl, (Object *) retval, OM_DISPOSE);
                    retval = NULL;
                }
            }
            break;

        /* Let the superclass handle everything else */
        default:
            retval = (ULONG) DoSuperMethodA (cl, o, msg);
            break;
    }

    return (retval);
}
```

Obtaining a Handle to the Source Data

The first step that a class has to perform in order to convert the source data, is to get a handle on the data. This is obtained by passing the DTA_Handle tag to the super-class using the OM_GET method. For example:

```
Object *o;
BPTR fh;

GetDTAttr (o, DTA_Handle, &fh, TAG_DONE);
```

If the source Type is DTF_IFF, then DTA_Handle points to a struct IFFHandle that is already initialized and opened for reading, otherwise the handle points to a BPTR file handle.

Handling Errors

Whenever an error occurs and the class is unable to continue converting the data, then it must set an appropriate error using the DOS SetIoErr() function and the <dos/dos.h> error codes or the error codes defined in <datatypes/datatypes.h>.

For example, if the class is unable to allocate memory:

```
if (buf = AllocVec (size, MEMF_CLEAR))
{
    ... continue with conversion ...
}
else
{
    SetIoErr (ERROR_NO_FREE_STORE);
}
```

Data Format

The following sections outline the required data format for each of the main object types.

Picture Class

The picture class is the super-class for any static graphic classes. The structures and tags used by this class are defined in <datatypes/pictureclass.h>.

A picture sub-class must fill out a BitMapHeader structure as well as provide a Mode ID, a BitMap and a ColorMap during the OM_NEW method of the class. There is no need to provide any other methods, as the remainder is handled by the picture class itself.

The picture sub-class needs to fill in any fields of the BitMapHeader structure that the class has data for. This will ensure that the picture data can then be saved to a file or copied to the clipboard.


```
struct BitMapHeader *bmh;
```

```
/* Obtain a pointer to the BitMapHeader structure from the picture class */
if (GetDTAttrs (dto, PDTA_BitMapHeader, &bmh, TAG_DONE) && bmh)
{
    /* Fill in some fields */
    bmh->bmh_Width = 640;
    bmh->bmh_Height = 200;
    bmh->bmh_Depth = 2;
}
```

Before manipulating any color information, the picture sub-class must first tell the picture class how many colors it has so that the information required for color remapping can be established.

```
SetDTAttrs (dto, PDTA_NumColors, ncolors, TAG_DONE);
```

After the number of colors have been established, the palette information can be filled in for the picture. The following fragment illustrates filling in the palette information.

```
struct ColorRegister *cmap;
WORD n, ncolors;
LONG *cregs;

/* Get a pointer to the color registers that need to be filled in */
GetDTAttrs (dto,
            PDTA_ColorRegisters, &cmap,
            PDTA_CRegs,          &cregs,
            TAG_DONE);

/* Set the color information */
for (n = 0; n < ncolors; n++)
{
    if (Read (fh, &rgb, QSIZE) == QSIZE)
    {
        /* Set the master color table */
        cmap->red    = rgb.rgbRed;
        cmap->green  = rgb.rgbGreen;
        cmap->blue   = rgb.rgbBlue;
        cmap++;

        /* Set the color table used for remapping */
        cregs[n * 3 + 0] = rgb.rgbRed << 24;
        cregs[n * 3 + 1] = rgb.rgbGreen << 24;
        cregs[n * 3 + 2] = rgb.rgbBlue << 24;
    }
    else
    {
        /* Indicate that we encountered an error. DOS Read
         * will have already filled in the IoErr() value */
        return (FALSE);
    }
}
```

The picture class must get the actual picture information in standard Amiga bitmap format. If the bitmap is allocated using the graphics AllocBitMap() function, then the picture class can dispose of the bitmap at OM_DISPOSE time.

```
struct BitMap *bm;

if (bm = AllocBitMap (bmh->bmh_Width, bmh->bmh_Height, bmh->bmh_Depth, BMF_CLEAR,
                     NULL))
{
    /* Tell the picture class about the picture data */
    SetDTAttrsA (dto, PDTA_BitMap, bm, TAG_DONE);
}
else
{
    /* Indicate the error and that we encountered an error */
    SetIoErr (ERROR_NO_FREE_STORE);
    return (FALSE);
}
```

A picture sub-class needs to provide the following fields to the super-class, during the OM_NEW method:

DTA_NominalHoriz (LONG)

Set to the width of the picture.

DTA_NominalVert (LONG)

Set to the height of the picture.

PDTA_ModeID (LONG)

A valid mode ID for the BitMap.

DTA_ObjName (STRPTR)

The name, or title, of the picture

DTA_ObjAuthor (STRPTR)

The author of the picture.

DTA_ObjAnnotation (STRPTR)

Notes on the picture.

DTA_ObjCopyright (STRPTR)

Copyright notice for the picture.

DTA_ObjVersion (STRPTR)

Version of the picture.

If a picture sub-class uses something other than AllocBitMap() to allocate the bitmap, then it must free the bitmap itself. This is done by implementing an OM_DISPOSE method for the sub-class.

```

ULONG ASM Dispatch (REG (a0) Class *cl, REG (a2) Object *o, REG (a1) Msg msg)
{
    struct ClassBase *cb = (struct ClassBase *) cl->cl_UserData;
    struct localData *lod;
    ULONG retval;

    switch (msg->MethodID)
    {
        case OM_NEW:
            /* ... */
            break;

        case OM_DISPOSE:
            /* Get a pointer to our object data */
            lod = INST_DATA (cl, o);

            /* Tell the picture class that it doesn't have a
             * bitmap to free any more */
            SetDTAttrs (o, PDTA_BitMap, NULL, TAG_DONE);

            /* Free the bitmap ourself */
            myfreebitmap (lod->lod_BitMap);

            /* Let the superclass handle everything else */
            default:
                retval = (ULONG) DoSuperMethodA (cl, o, msg);
                break;
    }

    return (retval);
}

```

Sound Class

The sound class is the super-class for any sampled audio classes. The structures and tags used by this class are defined in <datatypes/soundclass.h>.

A sound sub-class needs to provide the following fields to the super-class, during the OM_NEW method:

SDTA_Sample (UBYTE *)

8-bit sound data. The sound class will FreeVec() the sample data.

SDTA_SampleLength (ULONG)

Number of 8-bit bytes in the sound data.

SDTA_Volume (UWORD)

Number ranging from 0 being the quietest to 64 being the loudest.

SDTA_Period (UWORD)

Amount of time to play the sound.

SDTA_Cycles (UWORD)

Number of times to play the sound. 0 being an infinite loop.

DTA_ObjName (STRPTR)

The name, or title, of the sound

DTA_ObjAuthor (STRPTR)

The author of the sound.

DTA_ObjAnnotation (STRPTR)

Notes on the sound.

DTA_ObjCopyright (STRPTR)

Copyright notice for the sound.

DTA_ObjVersion (STRPTR)

Version of the sound.

No methods other than OM_NEW are needed, as the remainder is handled by the sound class itself.

The sound sub-class also needs to fill in any fields of the VoiceHeader structure that the class has data for. This will ensure that the sound data can then be saved to a file or copied to the clipboard.

```
struct VoiceHeader *vh;

/* Obtain a pointer to the VoiceHeader structure from the sound class */
if (GetDTAttrs (dto, SDTA_VoiceHeader, &vh, TAG_DONE) && vh)
{
    /* Fill in some fields */
    vh->vh_Octaves      = 1;
    vh->vh_Compression  = 0;
    vh->vh_Volume       = 63;
}
```

If a sound sub-class uses something other than AllocVec() to allocate the sound data, then it must free the sample itself. This is done by implementing an OM_DISPOSE method for the sub-class.

```
ULONG ASM Dispatch (REG (a0) Class *cl, REG (a2) Object *o, REG (a1) Msg msg)
{
    struct ClassBase *cb = (struct ClassBase *) cl->cl_UserData;
    struct localData *lod;
    ULONG retval;

    switch (msg->MethodID)
    {
        case OM_NEW:
```

```

        /* ... */
        break;

    case OM_DISPOSE:
        /* Get a pointer to our object data */
        lod = INST_DATA (cl, o);

        /* Tell the sound class that it doesn't have a
         * sample to free any more */
        SetDTAttrs (o, SDTA_Sample, NULL, TAG_DONE);

        /* Free the sample ourself */
        FreeMem (lod->lod_Sample, lod->lod_SampleLength);

        /* Let the superclass handle everything else */
        default:
            retval = (ULONG) DoSuperMethodA (cl, o, msg);
            break;
    }

    return (retval);
}

```

Text Class

The text class is the super-class for any formatted or non-formatted text classes. The structures and tags used by this class are defined in <datatypes/textclass.h>.

The text class provides the sub-class with a buffer that contains the text data. The sub-class must then provide the text class with a list of Line segments during the layout method. This Line list should only be created at gpl_Initial time, unless the TDTA_WordWrap attribute is TRUE.

```

struct Line {
    struct MinNode    ln_Link;
    STRPTR           ln_Text;
    ULONG             ln_TextLen;
    UWORD             ln_XOffset;
    UWORD             ln_YOffset;
    UWORD             ln_Width;
    UWORD             ln_Height;
    UWORD             ln_Flags;
    BYTE             ln_FgPen;
    BYTE             ln_BgPen;
    ULONG             ln_Style;
    APTR             ln_Data;
};

```

The Line structure fields are as follows:

ln_Link

MinNode used to link to the line list.

ln_Text

Pointer to the text for this line segment.

ln_TextLen

Number of bytes of text in this line segment.

ln_XOffset

Left pixel offset from the left edge of the object for this line segment.

ln_YOffset

Top pixel offset from the top edge of the object for this line segment.

ln_Width

Width of the line segment in pixels.

ln_Height

Height of the line segment in pixels.

ln_Flags

Control flags for this line segment.

LNF_LF

Used to indicate that this segment is the end of a line.

ln_FgPen

Pen to use for the foreground (the text color) for this line segment.

ln_BgPen

Pen to use for the background color for this line segment.

ln_Style

Text attribute soft style to use for this line segment.

As each Line segment is allocated it must be added to the Line list.

```
struct List *linelist;
struct Line *line;

/* Get a pointer to the line list */
if (GetDTAttrs (o, TDTA_LineList, (ULONG) &linelist, TAG_DONE) && linelist)
{
    /* Create a Line segment */
    if (line = AllocVec (sizeof (struct Line), MEMF_CLEAR))
    {
        /* Add it to the list */
        AddTail (linelist, (struct Node *)&line->ln_Link);
    }
}
```

Currently, this is the hardest class to sub-class, due to the number of methods that must be implemented. Following is the shell for a dispatcher and the layout method for a text sub-class.

```

struct localData
{
    VOID      *lod_Pool;
    ULONG     lod_Flags;
};

ULONG ASM Dispatch (REG (a0) Class * cl, REG (a2) Object * o, REG (a1) Msg msg)
{
    struct ClassBase *cb = (struct ClassBase *) cl->cl_UserData;
    struct localData *lod;
    struct List *linelist;
    ULONG retval = 0L;

    switch (msg->MethodID)
    {
        case OM_NEW:
            if (retval = DoSuperMethodA (cl, o, msg))
            {
                ULONG len, estlines, poolsize;
                BOOL success = FALSE;
                STRPTR buffer;

                /* Get a pointer to the object data */
                lod = INST_DATA (cl, (Object *) retval);

                /* Get the attributes that we need to determine
                 * memory pool size */
                GetDTAAttrs ((Object *) retval,
                            TDTA_Buffer,          (ULONG)&buffer,
                            TDTA_BufferLen,        (ULONG)&len,
                            TAG_DONE);

                /* Make sure we have a text buffer */
                if (buffer && len)
                {
                    /* Estimate the pool size that we will need */
                    estlines = (len / 80) + 1;
                    estlines = (estlines > 200) ? 200 : estlines;
                    poolsize = sizeof (struct Line) * estlines;

                    /* Create a memory pool for the line list */
                    if (lod->lod_Pool = CreatePool (MEMF_CLEAR | MEMF_PUBLIC,
                                                    poolsize, poolsize))
                    {
                        success = TRUE;
                    }
                    else
                    {
                        SetIoErr (ERROR_NO_FREE_STORE);
                    }
                }
            }
            else
            {
                {

```

```

        /* Indicate that something was missing that we
        * needed */
        SetIoErr (ERROR_REQUIRED_ARG_MISSING);
    }

    if (!success)
    {
        CoerceMethod (cl, (Object *) retval, OM_DISPOSE);
        retval = NULL;
    }
}
break;

case OM_UPDATE:
case OM_SET:
    /* Pass the attributes to the text class and force a refresh
    * if we need it */
    if ((retval = DoSuperMethodA (cl, o, msg)) && (OCLASS (o) == cl))
    {
        struct RastPort *rp;

        /* Get a pointer to the rastport */
        if (rp = ObtainGIRPort (((struct opSet *) msg)->ops_GInfo))
        {
            struct gpRender gpr;

            /* Force a redraw */
            gpr.MethodID = GM_RENDER;
            gpr.gpr_GInfo = ((struct opSet *) msg)->ops_GInfo;
            gpr.gpr_RPort = rp;
            gpr.gpr_Redraw = GREDRAW_UPDATE;
            DoMethodA (o, &gpr);

            /* Release the temporary rastport */
            ReleaseGIRPort (rp);
        }
        retval = 0;
    }
    break;

case GM_LAYOUT:
    /* Tell everyone that we are busy doing things */
    notifyAttrChanges (o, ((struct gpLayout *) msg)->gpl_GInfo, NULL,
                       GA_ID,      G(o)->GadgetID,
                       DTA_Busy,    TRUE,
                       TAG_DONE);

    /* Let the super-class partake */
    retval = (ULONG) DoSuperMethodA (cl, o, msg);

    /* We need to do this one asynchronously */
    retval += DoAsyncLayout (o, (struct gpLayout *) msg);
    break;

```



```

case DTM_PROCLAYOUT:
    /* Tell everyone that we are busy doing things */
    notifyAttrChanges (o, ((struct gpLayout *) msg)->gpl_GInfo, NULL,
                       GA_ID,      G(o)->GadgetID,
                       DTA_Busy,    TRUE,
                       TAG_DONE);

    /* Let the super-class partake and then fall through to our layout
     * method */
    retval = (ULONG) DoSuperMethodA (cl, o, msg);

case DTM_ASYNCCLAYOUT:
    /* Layout the text */
    retval = layoutMethod (cb, cl, o, (struct gpLayout *) msg);
    break;

case OM_DISPOSE:
    /* Get a pointer to our object data */
    lod = INST_DATA (cl, o);

    /* Don't let the super class free the line list */
    if (GetDTAttrs (o, TDTA_LineList, (ULONG) &linelist, TAG_DONE) &&
        linelist)
        NewList (linelist);

    /* Delete the line pool */
    DeletePool (lod->lod_Pool);

    /* Let the superclass handle everything else */
    default:
        retval = (ULONG) DoSuperMethodA (cl, o, msg);
        break;
}

return (retval);
}

ULONG layoutMethod (struct ClassBase *cb, Class * cl, Object * o, struct gpLayout *
gpl)
{
    struct DTSpecialInfo *si = (struct DTSpecialInfo *) G (o)->SpecialInfo;
    struct localData *lod = INST_DATA (cl, o);
    ULONG visible = 0, total = 0;
    struct RastPort trp;
    ULONG hunit = 1;
    ULONG bsig = 0;

    /* Switches */
    BOOL linefeed = FALSE;
    BOOL newseg = FALSE;
    BOOL abort = FALSE;

    /* Attributes obtained from super-class */
    struct TextAttr *tattr;

```

```

struct TextFont *font;
struct List *linelist;
struct IBox *domain;
ULONG wrap = FALSE;
ULONG bufferlen;
STRPTR buffer;
STRPTR title;

/* Line information */
ULONG num, offset, swidth;
ULONG anchor, newanchor;
ULONG style = FS_NORMAL;
struct Line *line;
ULONG yoffset = 0;
UBYTE fgpen = 1;
UBYTE bgpen = 0;
ULONG tabspace;
ULONG numtabs;
ULONG i, j;

ULONG nomwidth, nomheight;

/* Get all the attributes that we are going to need for a successful layout */
if (GetDTAttr (o,
               DTA_TextAttr, (ULONG) &tattr,
               DTA_TextFont, (ULONG) &font,
               DTA_Domain,   (ULONG) &domain,
               DTA_ObjName,  (ULONG) &title,
               TDTA_Buffer,  (ULONG) &buffer,
               TDTA_BufferLen, (ULONG) &bufferlen,
               TDTA_LineList, (ULONG) &linelist,
               TDTA_WordWrap, (ULONG) &wrap,
               TAG_DONE) == 8)
{
    /* Lock the global object data so that nobody else can manipulate it */
    ObtainSemaphore (&(si->si_Lock));

    /* Make sure we have a buffer */
    if (buffer)
    {
        /* Initialize the temporary RastPort */
        InitRastPort (&trp);
        SetFont (&trp, font);

        /* Calculate the nominal size */
        nomheight = (ULONG) (24 * font->tf_YSize);
        nomwidth  = (ULONG) (80 * font->tf_XSize);

        /* Calculate the tab space */
        tabspace = font->tf_XSize * 8;

        /* We only need to perform layout if we are doing word wrap, or this
         * is the initial layout call */
        if (wrap || gpl->gpl_Initial)

```

```

{
    /* Delete the old line list */
    while (line = (struct Line *) RemHead (linelist))
        FreePooled (lod->lod_Pool, line, sizeof (struct Line));

    /* Step through the text buffer */
    for (i = offset = num = numtabs = 0;
        (i <= bufferlen) && (bsig == 0) && !abort;
        i++)
    {
        /* Check for end of line */
        if (buffer[i]==13 && buffer[i+1]==10)
        {
            newseg = linefeed = TRUE;
            newanchor = i + 2;
            i++;
        }
        /* Check for end of page */
        else if (buffer[i] == 12)
        {
            newseg = linefeed = TRUE;
            newanchor = i + 1;
        }
        /* Check for tab */
        else if (buffer[i] == 9)
        {
            /* See if we need to terminate a line segment */
            if ((numtabs == 0) && num)
                newseg = TRUE;
            numtabs++;
        }
        else
        {
            /* See if we have any TABs that we need to finish out */
            if (numtabs)
            {
                offset += (((offset / tabspace) + 1) * tabspace) - offset;
                num = numtabs = 0;
                anchor = i;
            }

            /* Compute the width of the line. */
            swidth = TextLength (&trp, &buffer[anchor], num+1);
            if (offset + swidth > domain->Width)
            {
                /* Search for a whitespace character */
                for (j = i; (j >= anchor) && !newseg; j--)
                {
                    if (buffer[j] == ' ')
                    {
                        num -= (i - j);
                        newseg = TRUE;
                        i = j + 1;
                    }
                }
            }
        }
    }
}

```

```

    }

    newseg = linefeed = TRUE;
    newanchor = 1;
    i--;
}
else
{
    num++;
}
}

/* Time for a new text segment yet? */
if (newseg)
{
    /* Allocate a new line segment from our memory pool */
    if (line = AllocPooled (lod->lod_Pool, sizeof (struct Line)))
    {
        swidth = TextLength (&trp, &buffer[anchor], num);
        line->ln_Text      = &buffer[anchor];
        line->ln_TextLen   = num;
        line->ln_XOffset   = offset;
        line->ln_YOffset   = yoffset + font->tf_Baseline;
        line->ln_Width     = swidth;
        line->ln_Height    = font->tf_YSize;
        line->ln_Flags     = (linefeed) ? LNF_LF : NULL;
        line->ln_FgPen     = fgpen;
        line->ln_BgPen     = bgpen;
        line->ln_Style     = style;
        line->ln_Data      = NULL;

        /* Add the line to the list */
        AddTail (linelist, (struct Node *) line);

        /* Increment the line count */
        if (linefeed)
        {
            yoffset += font->tf_YSize;
            offset = 0;
            total++;
        }
        else
        {
            /* Increment the offset */
            offset += swidth;
        }
    }
    else
    {
        abort = TRUE;
    }

    /* Clear the variables */
    newseg = linefeed = FALSE;
}

```

```

        anchor = newanchor;
        num = 0;

        /* Check to see if layout has been aborted */
        bsig = CheckSignal (SIGBREAKF_CTRL_C);
    }
}
else
{
    /* No layout to perform */
    total = si->si_TotVert;
}

/* Compute the lines and columns type information */
si->si_VertUnit = font->tf_YSize;
si->si_VisVert = visible = domain->Height / si->si_VertUnit;
si->si_TotVert = total;

si->si_HorizUnit = hunit = 1;
si->si_VisHoriz = (LONG) domain->Width / hunit;
si->si_TotHoriz = domain->Width;

/* Release the global data lock */
ReleaseSemaphore (&si->si_Lock);

/* Were we aborted? */
if (bsig == 0)
{
    /* Not aborted, so tell the world of our newest attributes */
    notifyAttrChanges (o, gpl->gpl_GInfo, NULL,
                      GA_ID, G(o)->GadgetID,

                      DTA_VisibleVert, visible,
                      DTA_TotalVert, total,
                      DTA_NominalVert, nomheight,
                      DTA_VertUnit, font->tf_YSize,

                      DTA_VisibleHoriz, (ULONG) (domain->Width / hunit),
                      DTA_TotalHoriz, domain->Width,
                      DTA_NominalHoriz, nomwidth,
                      DTA_HorizUnit, hunit,

                      DTA_Title, title,
                      DTA_Busy, FALSE,
                      DTA_Sync, TRUE,
                      TAG_DONE);
}
}
return (total);
}

```

Defining a DataType Descriptor

DataTypees uses a simple descriptor to determine what type of data a file contains and what class, if any, is used to handle that data.

The DTDesc utility is used to define a DataType descriptor. The following steps describe how to use this utility to define a descriptor.

1. Load several sample files of the type being defined. This can be done by dropping their icons into the DTDesc window or by using the "Extras/Load Samples..." menu item.
2. The view area in the bottom right side of the DTDesc window will show the first 64 characters of the files. This area is used to define the Mask. The characters that don't match will be blotted out and only the similar characters will be shown. If more characters are shown as similar than really are, then they can easily be blotted out by rubbing over them.
3. DataTypes can be broken out into several different categories. Use the Group menu to select the category that the DataType descriptor belongs in.
4. The remaining fields must be filled out. Following is a table describing the fields and the information they require.

Name	Description
File Type	User description of the DataType.
Base Name	The base name of the DataType. The class library name is derived from this.
Name Pattern	The name of the file can be used to indicate the DataType. AmigaDOS wildcards can be used to specify the file name.
Function	A function can be used to further define a data type. This function must be a stand-alone executable that uses the DataType Descriptor Function Interface.
Case Sensitive?	The Mask can be either case sensitive or not.
Priority	Descriptors are sorted by Type, Function, NamePattern, and Mask. The priority field allows a DataType descriptor to be assigned a different priority than a similar DataType descriptor.
Type	This is a read-only field and is used to indicate what basic type a DataType is. Types include IFF, Binary and ASCII.

5. Once a DataType descriptor has been defined, it must be saved. Select the "Project/SaveAs..." menu item for the file requester used to save a DataType descriptor. The default name for a DataType descriptor is the text entered into File Type field.
6. In order for a new DataType descriptor to be loaded, the datatypes.library must be flushed from the system. This can either be done with the FlushLibs command or by using Avail Flush several times. Another way that the new DataType descriptor can be loaded is by using the AddDataTypes command with the REFRESH option.

DataType Descriptor Function Interface

Sometimes the fields within the DTDesc utility are not enough to define a DataType descriptor. In this case it is necessary to use a Function to further narrow down a DataType.

A Function is a stand-alone executable that expects the following arguments.

```
retval = Function (dthc);
d0                      a0
```

```
BOOL Function (struct DTHookContext *);
```

The function must return TRUE if the data matches, FALSE if it doesn't.

The DTHookContext structure contains the fields that are necessary for narrowing down the DataType. Following is a listing of the DTHookContext structure.

```
struct DTHookContext {
struct Library      *dthc_SysBase;
struct Library      *dthc_DOSBase;
struct Library      *dthc_IFFParseBase;
struct Library      *dthc_UtilityBase;

/* File context */
BPTR                dthc_Lock;          /* Lock on the file */
struct FileInfoBlock *dthc_FIB;         /* Pointer to a FileInfoBlock */
BPTR                dthc_FileHandle; /* Pointer to the file handle (may be NULL) */
struct IFFHandle     *dthc_IFF;         /* Pointer to an IFFHandle (may be NULL) */
STRPTR              dthc_Buffer;        /* Buffer */
ULONG               dthc_BufferLength; /* Length of the buffer */
};
```

The DTHookContext structure fields are as follows:

dthc_SysBase dthc_DOSBase dthc_IFFParseBase dthc_UtilityBase

These are library bases that your function can utilize. It will need to open any other libraries that it needs.

dthc_Lock

If the source data is a DOS file, then this is a lock on the file.

dthc_FIB

If the source data is a DOS file, then this is a filled in FileInfoBlock for the file.

dthc_FileHandle

If the source data is a DOS file, then this is the file handle for the file otherwise the handle will be NULL. The file is guaranteed to be at the beginning.

dthc_IFF

If the source data is IFF, then this is the IFFHandle for accessing the data, otherwise the handle will be NULL. The position is guaranteed to be at the beginning of the data. The DOS file fields must not be accessed if the data is IFF.

dthc_Buffer

This buffer contains the first dthc_BufferLength bytes of data.

dthc_BufferLength

Indicates the number of bytes in dthc_Buffer, up to 64.

The following example shows how to write a simple DataTypes Descriptor Function.

```

/* This example is to be compiled with no stack checking.  It must not be linked
 * with any startup code so that DTHook is the entry point for the executable.
 */
#include <exec/types.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <datatypes/datatypes.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/utility_protos.h>

#include <pragmas/exec_pragmas.h>
#include <pragmas/dos_pragmas.h>
#include <pragmas/utility_pragmas.h>

/*****
#define SysBase      dthc->dthc_SysBase
#define DOSBase      dthc->dthc_DOSBase
#define UtilityBase  dthc->dthc_UtilityBase
*****/

BOOL __asm DTHook (register __a0 struct DTHookContext * dthc)
{
    BOOL retval = FALSE;
    register ULONG i;
    UBYTE ch;

    /* Make sure we have a buffer */
    if (dthc->dthc_Buffer)
    {
        for (i = 0; (i < dthc->dthc_BufferLength) && !retval; i++)
        {
            ch = dthc->dthc_Buffer[i];

            /* Look at the data... */
        }
    }
    return retval;
}

```

The next example shows how to write a DataTypes Descriptor that looks into an IFF file for needed chunk information.

```

/* This example is to be compiled with no stack checking.  It must not be linked
 * with any startup code so that DTHook is the entry point for the executable.
 */
#include <exec/types.h>
#include <dos/dos.h>
#include <dos/dosextens.h>

```

```

#include <datatypes/datatypes.h>
#include <datatypes/pictureclass.h>
#include <libraries/iffparse.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>
#include <clib/utility_protos.h>

#include <pragmas/exec_pragmas.h>
#include <pragmas/dos_pragmas.h>1
#include <pragmas/iffparse_pragmas.h>
#include <pragmas/utility_pragmas.h>

/*****/
#define SysBase      dthc->dthc_SysBase
#define DOSBase      dthc->dthc_DOSBase
#define IFFParseBase dthc->dthc_IFFParseBase
#define UtilityBase  dthc->dthc_UtilityBase
/*****/
#define BMH_SIZE (sizeof (struct BitMapHeader))
/*****/

BOOL __asm DTHook (register __a0 struct DTHookContext * dthc)
{
    struct BitMapHeader bmh;
    struct ContextNode *cn;
    struct IFFHandle *iff;
    BOOL retval = FALSE;

    /* Make sure that this is an IFF data type */
    if (iff = dthc->dthc_IFF)

        /* Stop on the BitMapHeader (type, id) */
        if (StopChunk (iff, ID_ILBM, ID_BMHD) == 0)

            /* Scan through the IFF handle */
            if (ParseIFF (iff, IFFPARSE_SCAN) == 0L)

                /* Make sure we have a current chunk */
                if (cn = CurrentChunk (iff))

                    /* Make sure the current chunk is ILBM BMHD */
                    if (((cn->cn_Type == ID_ILBM) && (cn->cn_ID == ID_BMHD)))

                        /* Read the chunk data */
                        if (ReadChunkBytes (iff, &bmh, BMH_SIZE) == BMH_SIZE)

                            /* See if the depth is set to 24 */
                            if (bmh.bmh_Depth == 24)
                                retval = TRUE;

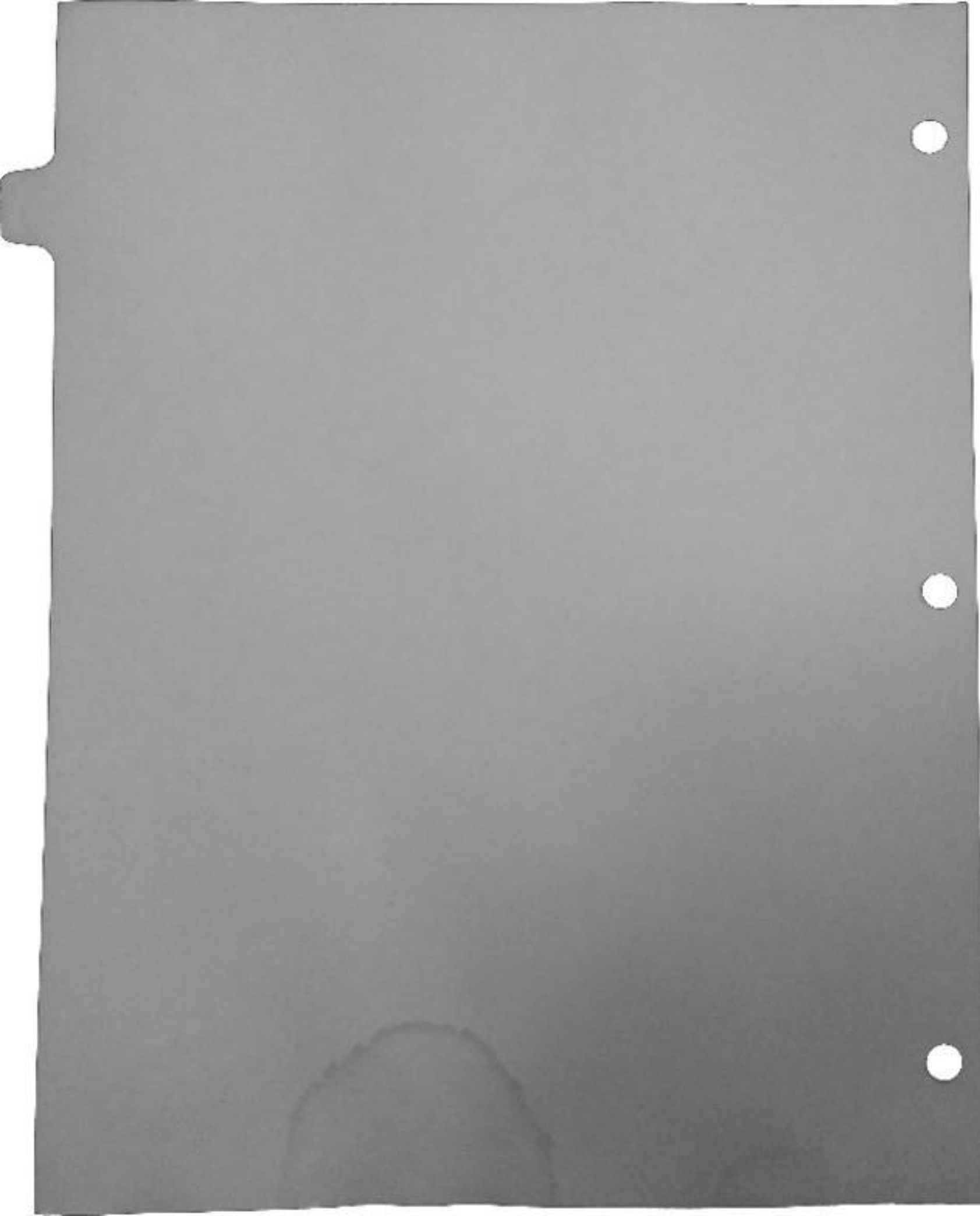
    return (retval);
}

```

C

C

C



Changes in the V39 OS

by Martin Taillefer

Many changes were made to the system software for V39. The major differences are in the graphics subsystem and in Intuition. There were also hundreds of changes made to other areas of the system software. This document presents the majority of these changes, and explains the possible impact of these new features on the developer.

V39 requires both a new disk set, and a new ROM. V38 is a disk-based upgrade requiring only a new disk set. For a complete list of changes in V38, refer to "Release 2.1 Overview" in the *V39 Release Notes*, available from CATS (part number: V3.0).

The main changes to the contents of the disks in V39 are detailed in Appendix A and in Appendix B. Here are the highlights of changes to system disk organization:

- ☐ The V38/V39 system software is shipped on FFS floppies. This is mainly to increase available storage.
- ☐ Under V38, a Storage drawer was created on the Extras disk. Under V39, a separate Storage disk is provided. Storage contains the same five drawers as in the Devs drawer. This is where things that are not currently used are kept. To activate a monitor, the user is expected to drag the desired monitor icon from Storage/Monitors to Devs/Monitors. Same applies for printer drivers, keymaps, DOS drivers, and datatypes
- ☐ The Fonts disk is now called FONTS instead of AmigaFonts. This enables the disk to be used directly whenever fonts are needed by the system, instead of requiring the user to assign FONTS: to the disk.
- ☐ Note that there are different specific disk sets shipped with different machines and packages. For example, low-end V38 systems do not get a Locale disk, and instead have a Locale directory on the Workbench disk.
- ☐ Deleted Files. The following system files present under V37 are no longer included with the system software.

Devs/narrator.device	This device is deleted as part of the removal of speech support.
Libs/translator.library	This library is deleted as part of the removal of speech support.

Tools/Colors	This tool has been removed in V39 because its functionality conflicted with pen sharing, and was generally not very nice to applications.
Utilities/Display	This program is replaced under V39 by the more flexible and powerful MultiView utility.
Utilities/More.info	This icon has been removed because we want to encourage users to use MultiView instead of More when running from Workbench. The More program itself remains on the disk since it is referenced by many read me files, and the like.

- ❑ **Speech Support.** The various components supporting synthesized speech are no longer included with the operating system. This can be a serious liability for applications depending on synthesized speech for correct operation. The V37 components supporting speech still function correctly under V38 and V39. The V38/V39 installation procedures do not remove the V37 files when updating a system.

Further details of how the system disks have changed can be found in Appendix A and B. V39 ROM changes are listed in Appendix C and D.

V38/V39 API Changes

This section discusses what changes in the programming model can affect developers.

Finding Version Information

An important point to mention is how to determine if a system is running V38 instead of V37. The recommended approach is to open `version.library`, and check its version. For example:

```
struct Library *VersionBase;
if (VersionBase = OpenLibrary("version.library",0))
{
    if (VersionBase->lib_Version >= 38)
    {
        /* user is running at least V38 Workbench */
    }
    else
    {
        /* user is running at most V37 Workbench */
    }
}
else
{
    /* can't tell what the user is running, assume the minimum version
    * that your application supports */
}
```

The above technique lets you determine which general version is in use for the disk-based software. *Never* assume this is a reflection of every other library in the system. For example, if you need features that are only present in V38 `asl.library`, you must explicitly check the version of `asl.library` before using it. The same is true for all other system libraries.

To determine the general version of the ROM, use `SysBase->LibNode.lib_Version`.

GadTools Library

Many enhancements were made to GadTools for V39. Below you will find short descriptions of some of the new features. A complete list of all the new features is provided in Appendix A, and more details can be had in gadtools.doc.

Menus. GadTools fully supports Intuition's NewLook menus. NewLook menus can be added to an application by providing the {WA_NewLookMenus, TRUE} tag to OpenWindowTags(), and providing the {GTMN_NewLookMenus, TRUE} tag to LayoutMenus(). These two tags will give your application NewLook menus. The pens used to render these menus are controllable by the user through the V39 Palette prefs editor.

You can now put an arbitrary command string in the right-hand side of a menu, where the Amiga-key equivalent normally goes. To do this, point the NewMenu.nm_CommKey field at the string (e.g., "Shift-Alt-F1), and set the new NM_COMMANDSTRING flag in NewMenu.nm_Flags.

GT_GetGadgetAttrs(). GadTools now has a GT_GetGadgetAttrsA() function which retrieves attributes of the specified gadget, according to the attributes chosen in the tag list. For each entry in the tag list, ti_Tag identifies the attribute, and ti_Data is a pointer to the long variable where you wish the result to be stored.

Here is an example:

```
LONG top = 0;
LONG selected = 0;
LONG result;
result = GT_GetGadgetAttrs(listview_gad, win, NULL,
                           GTLV_Top, &top,
                           GTLV_Selected, &selected,
                           TAG_DONE);

if (result != 2)
{
    printf( "Something's wrong!" );
}
```

Palettes. GadTools palette gadgets got a major overhaul for V39. There are a few new features, some reduction in memory usage, and an increase in performance.

Palette gadgets no longer display a box filled with the selected color. The selected color is instead denoted by a box drawn around the color square in the main palette area. This change slightly impacts the size of palette gadgets. Depending on the conditions, V39 PALETTE_KIND gadgets can be slightly smaller than the V37 ones.

Another change which can affect the size of the palette gadgets is the smarter layout of the color squares. An attempt is now made to keep the color squares as square as possible, based on the aspect ratio information obtained from the gfx database. As many color squares as possible are put on the screen, until things get too small in which case the upper colors are thrown away.

The `GTPA_ColorTable` tag was added in support of sparse color tables. It lets you define arbitrary pens to be used to fill-in the color squares of the palette gadget.

The `GTPA_NumColors` tag lets you define the exact number of colors to display in the palette gadget. Under V37, the `GTPA_Depth` tag played the same role. The problem with `GTPA_Depth` is that only multiples of 2 in number of colors can be specified. `GTPA_NumColors` allows any number of colors.

Listviews. GadTools listviews got a major overhaul for V39. There are a few new features, some reduction in memory usage, and an increase in performance.

The biggest difference that is readily noticeable is the disappearance of the display box at the bottom of the scrolling list area. Instead of showing the selected item in the display box, the selected item remains highlighted within the scrolling list. This was done to prepare listviews for multi-select support. Multi-selection requires a highlighting scheme as a display box would not work. The removal of the display box has a slight impact on the size of the listview. Listviews that had a display box under V37 got slightly smaller vertically under V39.

The other major addition to GadTools listviews is `GTLV_Callback`. This tag allows a callback hook to be provided to gadtools for listview handling. Currently, the hook only gets called to render an individual item. Every time GadTools wishes to render an item in a listview, it invokes the call back function, which can do custom rendering. This allows GadTools listviews to be used to scroll complex items such as graphics and images.

Finally, listviews can now be disabled using `{GA_Disabled, TRUE}`.

The ASL Library

ASL got rewritten for V38. Its file requester is now extremely fast and offers more features to the user, and to the programmer. Its font requester now comes up about twice as fast the first time it is displayed, and instantaneously for subsequent uses. ASL also now includes a screen mode requester, which is very useful in the context of AA, and will be invaluable for AAA and RTG.

New Names. V38 `<libraries/asl.h>` contains new names for most of the structures, tags, and flag bits used by ASL. The names are now consistent, and more descriptive. We encourage the use of the new names as much as possible.

To ensure your code only uses the new names, just define the symbol `ASL_V38_NAMES_ONLY` before including `<libraries/asl.h>`:

```
#define ASL_V38_NAMES_ONLY
#include <libraries/asl.h>
```

The symbol definition will prevent the old-style names from being defined. This is the same method used to disable the pre-V37 names in Intuition header files.

Don't Trust Me. The many filtering tags that you can supply to the ASL requesters are only advisory in nature. For example, even if you request that only non-proportional fonts be allowed in the font requester, the user can still type in the name of a proportional font. The user can also enter out-of-bounds values for any numeric parameter. You must check all results for sanity prior to using them.

Common ASL Tags. The tags for the different requesters are now distinct from one another instead of being all grouped together. All options that could be set via flag bits now have individual tags to control them. For example, there is now an ASLFR_DoSaveMode tag that does the same as the FRF_DOSAVEMODE flag bit.

Some basic tags are supported by all requester types:

ASLXX_Window. This indicates the parent window of the requester. You provide a window pointer, and if you don't specify an ASLXX_Screen tag, then the ASL requester will open on the same screen as this window.

ASLXX_PubScreenName. You provide this tag with the name of a public screen to open the ASL requester on.

ASLXX_Screen. You provide this tag with a pointer to a Screen to open on. If none of the above three tags are specified, then the ASL requester will open on the current default public screen. This will most likely be Workbench.

ASLXX_PrivateIDCMP. By default, ASL requesters use the same IDCMP port as their parent window (specified using ASLXX_Window). You can request that a private IDCMP port be used by setting this tag to TRUE. If you don't provide a parent window, then a private port is automatically used and this tag need not be provided.

ASLXX_IntuiMsgFunc. You provide this tag a pointer to a Hook structure, which indicates a routine to run whenever an IntuiMessage arrives at the ASL requester's IDCMP port that is not meant for the requester's window. This happens whenever the ASL requester is sharing the parent's window IDCMP port, and a message for the parent window arrives at the port. The function being called can process the message for the application.

ASLXX_SleepWindow. If you set this tag to TRUE, ASL will put the parent window (specified using ASLXX_Window) to sleep until the ASL requester is satisfied. This involves blocking all input in the parent window, and displaying a busy pointer in it.

ASLXX_TextAttr. You can provide a pointer to a standard TextAttr structure which defines

the font to use in the ASL requester. If this tag is not supplied, the default behavior is to use the font of the screen on which the requester opens. Note that just like for Intuition objects, the font indicated by the TextAttr structure must already be in memory before using the requester. There is no guarantee that ASL will actually use the font you asked for. If that font is too large, ASL will use a smaller one. Also, as of V39, the file requester's file list requires a monospace font. So if you provide a proportional font, the file requester will use the system default font instead for its list of files.

ASLXX_Locale. You pass this tag a pointer to a Locale structure, as obtained from the `locale.library/OpenLocale()` function. The locale is used to determine in which language, and using which country information, the various requesters should be displayed. If this tag is not provided, or its value is NULL, then the default system Locale is used. This corresponds to what the user has currently picked in the Locale prefs editor. As of V39, certain items such as the dates in the file requester always use the system default Locale instead of the Locale provided with this tag.

ASLXX_TitleText. Provide this tag a string which will be used for the title of the requester. If this tag is not provided, the requester has no title.

ASLXX_PositiveText. Provide this tag a string which will be used for the label of the positive choice gadget in the requester. If this string is not provided, a localized default is used (English default is "OK"). We recommend using this tag to make the action of the various requesters more clear to the user. For example, when a file requester is being displayed to load a file, it is clearer if the gadget says "Open" instead of simply "OK".

ASLXX_NegativeText. Provide this tag a string which will be used for the label of the negative choice gadget in the requester. If this string is not provided, a localized default is used (English default is "Cancel").

ASLXX_InitialLeftEdge, ASLXX_InitialTopEdge, ASLXX_InitialWidth,

ASLXX_InitialHeight. These four tags let you specify the position and size of the ASL requester window. These are only requests that ASL may decide to ignore. When an ASL requester is closed, it is easy to determine the position and size of the requester when the user closed it. It is a good idea for an application to remember these values and use them in subsequent invocations of the requester.

The File Requester. The file requester supports a number of tags. Some of the tags introduced in V38 are discussed here.

Note that the future holds many changes in the ASL file requester that will substantially increase its flexibility to both the programmer and the user. If you use the ASL file requester in your applications today, you will automatically get the majority of these new features when they become available. Rolling your own file requester means a lot of extra work, and means that you will not automatically benefit from the new interface when it becomes available.

ASLFR_DoSaveMode. Setting this tag to TRUE indicates the file requester is being used for saving information. When in save mode, the file requester changes slightly. The most obvious change is the use of a different set of colors in the file list. This is a direct queue to the user that something is being selected for writing, not reading. Second change is that if the user enters a directory name which doesn't exist, ASL will prompt the user to see if the user wishes to create the directory. This lets the user create new directories to save files into. There are likely to be more differences between normal and save mode in the future. Please use this tag when appropriate.

ASLFR_RejectIcons. When set to TRUE, this tag prevents icons (.info files) from being displayed in the file requester. If this tag is not specified, icons will be displayed by the file requester. Please use this tag in all your software. Workbench users should never have to see .info files. The default behavior of the file requester is to display .info files, which is incorrect. Unfortunately, this default behavior cannot be changed due to compatibility.

ASLFR_DoPatterns. Setting this tag to TRUE causes a Pattern gadget to be displayed in the file requester, which allows the user to enter AmigaDOS patterns to filter out files. The default is to have no pattern gadget.

ASLFR_DrawersOnly. Setting this tag to TRUE causes the file requester to have no File gadget, and to only display directory names in its file list. This is a useful option if you wish to have the user select a destination directory for a particular task.

ASLFR_RejectPattern. Provide an AmigaDOS pattern to this tag, and any files matching this pattern will not be displayed in the file requester. This pattern can never be edited by the user. Note that the pattern you provide here must have already been parsed by `dos.library/ParsePatternNoCase()`.

ASLFR_AcceptPattern. Provide an AmigaDOS pattern to this tag, and only files matching this pattern will be displayed in the file requester. This pattern can never be edited by the user. Note that the pattern you provide here must have already been parsed by `dos.library/ParsePatternNoCase()`.

ASLFR_FilterDrawers. Setting this tag to TRUE causes the **ASLFR_RejectPattern**, **ASLFR_AcceptPattern**, and the Pattern text gadget to also apply to filter drawer names. Drawers are normally never affected by pattern filtering.

The Screen Mode Requester. The screen mode requester allows the user to select amongst the wide range of display modes available on the Amiga. Future chip sets, and the RTG effort will yield a very large additional number of display modes. Using the ASL requester is a good way to ensure that you benefit from the maximum upwards compatibility possible. In many cases, by using the screen mode requester, your applications will automatically be able to open displays in new modes that are introduced in future OS revisions.

An important point to note is that the interface presented to the user by the screen mode requester is likely to change dramatically in the future. This is because the increased number of available modes will make the current interface difficult to use and understand at best. A new scheme is being developed that should allow the user to more easily and accurately pick display modes. If you use the ASL screen mode requester in your applications instead of rolling your own version, your application will automatically get this new interface when it becomes available.

The ultimate goal of the screen mode requester is to return a graphics mode id. The mode ids are used since V37 to distinguish between the different display modes in the system. The screen mode requester can also return additional information including requested display sizes and depth.

As all other ASL requesters, you get the result of a request by reading the contents of the requester structure after the `AslRequest()` calls return success. There are two fields worth mentioning in the `ScreenModeRequester` structure: `sm_BitMapWidth` and `sm_BitMapHeight`. These values define the values to pass to `AllocRaster()` and `InitBitMap()` when hand-building a `BitMap` structure to display the requested width, height, and depth. This is useful when running on a V37 ROM. For V39 use, simply use the values in `sm_Width` and `sm_Height`, and pass those to `AllocBitMap()` directly.

Here are the most important screen mode requester tags.

ASLSM_InitialDisplayID. This tag sets the initial mode that is selected in the mode listview. You provide it a graphics mode id, and that mode will be initially selected in the file requester.

ASLSM_InitialDisplayWidth, ASLSM_InitialDisplayHeight, ASLSM_InitialDisplayDepth. These tags determine the initial setting of three gadgets that can optionally be displayed in the requester. The default if these tags are not provided are a width of 640, a height of 200, and a depth of 2.

ASLSM_InitialOverscanType. This tag determines the initial setting of the Overscan Type cycle gadget. This optional gadget lets the user pick which overscan setting to use for the given mode. The values you can provide to this tag are: `OSCAN_TEXT`, `OSCAN_STANDARD`, `OSCAN_MAXIMUM`, and `OSCAN_VIDEO`. `OSCAN_VIDEO` is only available starting with V39, it is not in V38. The four `OSCAN_XX` constants are defined in `<intuition/screens.h>`.

ASLSM_InitialAutoScroll. This tag can be set to `TRUE` or `FALSE` and determines what the initial state of the option AutoScroll checkbox gadget should be.

ASLSM_DoWidth, ASLSM_DoHeight, ASLSM_DoDepth, ASLSM_DoOverscanType, ASLSM_DoAutoScroll. These tags control which of the optional gadgets is displayed in the screen mode requester. The default is to have none of these present. Setting any of these tags to TRUE will make the gadget appear.

ASLSM_MinWidth, ASLSM_MaxWidth, ASLSM_MinHeight, ASLSM_MaxHeight, ASLSM_MinDepth, ASLSM_MaxDepth. These tags let you specify limits to the values the user is allowed to enter in the requester.

ASLSM_PropertyFlags, ASLSM_PropertyMask. These two tags cooperate to allow a flexible means of filtering out unwanted display modes. Each display mode in the graphics database has a given set of properties associated with it. These tags allow you to determine which modes to display in the mode list of the requester based on the properties of the modes.

The mode properties are defined as a ULONG of flag bits. The definitions for all supported properties are in *<graphics/displayinfo.h>*. Examples include DIPF_IS_WB and DIPF_IS_LACE.

ASLSM_PropertyMask defines which properties you are concerned with. Those are the only bits for which the specific setting matters to you. The setting of all other property bits doesn't matter to you.

ASLSM_PropertyFlags defines exactly in which state the property flags you have shown interest in via the ASLSM_PropertyMask tag should be.

The way ASLSM_PropertyMask and ASLSM_PropertyFlags interact is identical to how the two flags parameters interact in *exec.library/SetSignal()*. This is how ASL uses the tag values internally:

```
if ((displayInfo.PropertyFlags & propertyMask) ==
    (propertyFlags & propertyMask))
{
    /* Mode accepted */
}
else
{
    /* Mode rejected */
}
```

An example can help understand the relationship between the two tags. If you want to display only screen modes that can be used for the Workbench screen, and that are not interlaced, you would set ASLSM_PropertyMask to (DIPF_IS_WB| DIPF_IS_LACE), and ASLSM_PropertyFlags to (DIPF_IS_WB). The mask value means you are interested in the state of the DIPF_IS_WB and DIPF_IS_LACE bits. The flags value says you want the DIPF_IS_WB to be set, and the DIPF_IS_LACE bit to be clear.

The Locale Library

The `locale.library` provides the core of system localization. It is explained in detail in the *V39 Release Notes* available from Commodore's CATS department (part number: V3.0). Also refer to `locale.doc` for more information.

The Bullet Library

The `bullet.library` contains the Compugraphic outline font rendering engine. It is explained in the *V39 Release Notes* available from Commodore's CATS department (part number: V3.0). Also refer to `bullet.doc` for more information.

Color Wheel and Gradient Slider

The color wheel and gradient slider are two new BOOPSI classes introduced in V39. Together, they offer a very nice interface to let the user select or adjust colors. The V39 Palette prefs uses both classes in its interface.

Using either of these classes requires you to first open them as libraries, and then create new BOOPSI objects using the `NewObject()` function:

```
if (ColorWheelBase = OpenLibrary("gadgets/colorwheel.gadget",0))
{
    cw = NewObject(...);
}
```

For an introduction to programming the color wheel and gradient slider refer to the *Amiga Mail* newsletter, Sept/Oct 1992 issue, page IV-91 also published in *V39 Release Notes* available from Commodore's CATS department (part number: V3.0).

The `colorwheel.gadget`. The color wheel class provides the ability to create gadgets enabling the user to control the hue and saturation components of an HSB (Hue-Saturation-Brightness) color space. The companion gradient slider class enables control of the brightness component of the color space.

The color wheel can operate on screens of any depth, and adapts its rendering to the number of colors available. The system's pen sharing system is used in order to maximize the number of colors used by the wheel. A color wheel gadget is (normally) responsible for choosing its own color pens to draw in (using `graphics.library/ObtainBestPen()`). However, the creator of the gadget can "donate" some of its pens to the gadget, using the `WHEEL_Donation` tag.

The reason that the color wheel picks its own colors is because it has the ability to display several different layouts depending on the number and variety of colors available. For example, when opening on a screen of low depth or when opening on a screen where all the

pens have already been allocated exclusively, the gadget will display a "monochrome" version of the color wheel, where instead of colored segments, the letters "R" (for red), "G" (for green), "B" (for blue), "Y" (for yellow), "C" (for cyan) and "M" (for magenta) will be used as labels.

You can talk to the color wheel using HSB or RGB, even though the color wheel only really deals with HSB in its user-interface. All communications with applications are performed with full 32-bit color component values. Internally everything is currently kept and processed in 16-bit space, although this might change in the future.

Here are the main tags supported by the color wheel class:

WHEEL_Hue. Lets you control the Hue component of the color wheel. This is effectively the angle around the wheel where the desired color lies. A hue value of 0 is all red, and nothing but red. Increasing the value moves the color towards all green at \$55555555, full blue at \$AAAAAAAA, and back to red at \$FFFFFFFF.

WHEEL_Saturation. Lets you control the Saturation component of the color wheel. This is effectively the distance from the center of the wheel where the desired color lies. A saturation value of 0 puts the knob at the center of the wheel and always yields white. Increasing the value progressively moves the knob farther away from the center. until the value \$FFFFFFFF is reached in which case the knob is as far as it can go.

WHEEL_Brightness. Lets you control the Brightness component of the color wheel. The color wheel does not itself have any means of displaying or editing brightness, but it does maintain this value internally. Used with the **WHEEL_GradientSlider** tag, you can control the value of a gradient slider object by passing **WHEEL_Brightness** to a color wheel. A brightness value of 0 means all black. Increasing the value progressively brightens the current color, until the value \$FFFFFFFF is reached in which case the color is as bright as it gets.

WHEEL_Red, WHEEL_Green, WHEEL_Blue. Let you specify new RGB values for the color wheel. The values provided are converted into HSB values are then used.

WHEEL_Screen. This is a required tag that lets you specify the screen on which the color wheel is to be displayed. Note that once a color wheel is created, the screen should not be closed until the color wheel object is discarded using **DisposeObject()**.

WHEEL_BevelBox. If you set this tag to **TRUE**, a bevel box will be drawn around the color wheel object.

WHEEL_GradientSlider. This tag lets you attach a gradient slider object to the color wheel. You give this tag a pointer to a gradient slider object obtained previously from **NewObject()**. Once this is done, anytime the various tags that can affect the brightness component of the current color is sent to the color wheel, the color wheel automatically

changes the value of the attached gradient slider to match that new brightness value. Reading the brightness value from the color wheel returns the current value indicated by the gradient slider.

Using this tag effectively allows you to treat the color wheel and gradient sliders as a single gadget. Once things are set up, all communications occur through the wheel object, and the gradient slider can pretty much be ignored.

The `gradientslider.gadget`. The gradient slider gadget class is a type of non-proportional slider. The primary feature of the gradient slider is it's appearance. Unlike normal sliders, a gradient slider can display a "spread of colors" or "color gradient" in the slider container box. The "knob" or "thumb" of the slider appears to slide on top of this color gradient.

The color gradient effect is built-up using a combination of multiple pens and half-tone dithering. The application must tell the slider exactly which pens to use in creating the gradient effect, and in what order to use them. Essentially, it does this by passing an array of pens (terminated by ~0, just like a `PenSpec`) to the slider. The first pen in the array is used as the color at the top of the slider (or left, if it is horizontal), and the last color in the array is used at the bottom (or right). The other pens will be used at evenly spaced intervals in between. Dithering is used to smoothly fade between the pens, allowing the illusion of a continuous change in color.

Here are the main tags supported by the gradient slider class:

GRAD_MaxVal. Lets you set the maximum value supported by the slider. This must be in the range \$0..\$FFFF.

GRAD_CurVal. Lets you set the current value of the slider. This should be in the range 0..GRAD_MaxVal.

GRAD_PenArray. Lets you specify an array of pens that the slider should use to create its gradient background. The array can contain any number of pens, and is terminated with a pen value of ~0. These pens can be allocated as shared, since their RGB value is not altered by the slider. The first pen is used on the top or left of the slider, and the last pen is used on the bottom or right. All other pens are evenly spaced out and used in between. Dithering is used between the pens to enhance the smoothness of the gradient transition.

A NULL pen array causes the background of the slider to be rendered in the screen's background color. A pen array containing only a single pen causes the background to be rendered using that pen.

New File Systems

The V39 ROM file system supports 6 different disk formats:

- ❑ **DOS0** This is the traditional Amiga file system. It has 488 bytes of data per block. It offers a high level of redundancy and validation which makes it very reliable. It is also fairly slow.
- ❑ **DOS1** This is the original Fast File System introduced in 1.3. It uses a disk layout quite similar to DOS0, except it forgoes some redundancy in the name of speed. It stores 512 bytes per block, and runs substantially faster than DOS0.
- ❑ **DOS2** This is an international flavor of DOS0. The only difference between this format and DOS0 is the hashing algorithm used to locate the files and directories on the disk. DOS0 has a bug in dealing with making international characters case-sensitive. DOS2 corrects this bug, and is otherwise identical to DOS0.
- ❑ **DOS3** This is an international flavor of DOS1. The only difference between this format and DOS1 is the hashing algorithm used to locate the files and directories on the disk. DOS1 has a bug in dealing with making international characters case-sensitive. DOS3 corrects this bug, and is otherwise identical to DOS1.
- ❑ **DOS4** This is a directory-caching version of DOS2. It has the same general disk layout, with the addition of special directory cache blocks. These cache blocks store the contents of each directory. The advantage of this is that getting a directory listing can be an order of magnitude faster than on normal DOS2 disks. The directory caches require a slight amount of disk space, and maintaining them slightly slows down file creation, deletion, and update. This file system is mostly meant for floppies, as it doesn't generate a very noticeable performance increase on hard drives.
- ❑ **DOS5** This is a directory-caching version of DOS3. It has the same general disk layout, with the addition of special directory cache blocks. These cache blocks store the contents of each directory. The advantage of this is that getting a directory listing can be an order of magnitude faster than on normal DOS3 disks. The directory caches require a slight amount of disk space, and maintaining them slightly slows down file creation, deletion, and update. This file system is mostly meant for floppies, as it doesn't generate a very noticeable performance increase on hard drives.

Disk Block Format. Here are some notes concerning changes to the disk block format required by the new SetOwner() call, and more importantly the changes present in DOS4 and DOS5 to support directory caching.

The format of root blocks is unchanged under V39.

The format of file header blocks is slightly different in order to store owner information. One of the reserved fields is now being used:

```
struct FileHeaderBlock
{
    ULONG   fhb_Type;           // T.SHORT
    ULONG   fhb_OwnKey;         // key of this block
    ULONG   fhb_HighSeq;        // total blocks in file
    ULONG   fhb_DataSize;       // number of data block slots used
    ULONG   fhb_FirstBlock;     // first block in this file
    ULONG   fhb_Checksum;       // checksum of this block

    ULONG   fhb_HashTable[];    // variable number of hash table entries

    ULONG   fhb_Reserved        // always 0
    ULONG   fhb_OwnerXID        // owner UID/GID
    ULONG   fhb_Protect         // protection bits
    LONG    fhb_ByteSize        // total size of file in bytes
    char    fhb_Comment[92];    // comment as a BCPL string
    ULONG   fhb_Days;           // creation date and time
    ULONG   fhb_Mins;
    ULOGN   fhb_Ticks;
    char    fhb_FileName[36];   // filename as BCPL string
    ULONG   fhb_Link;           // pointer to object header is linked to
    ULONG   fhb_BackLink;       // pointer to previous object in link chain
    ULONG   fhb_HashChain;      // next entry with same hash value
    ULONG   fhb_Parent;         // pointer back to parent directory
    ULONG   fhb_Extension;      // 0 or pointer to first extension block
    ULONG   fhb_SecType;        // ST.FILE
};
```

User directory blocks were modified to add the same owner information as for file header blocks. Under DOS4 and DOS5, the user directory blocks also point to the new directory cache blocks.

```
struct UserDirectoryBlock
{
    ULONG   udb_Type;           // T.SHORT
    ULONG   udb_OwnKey;         // key of this block
    ULONG   udb_SeqNum;         // highest seq (always 0)
    ULONG   udb_DataSize;       // always 0
    ULONG   udb_NextBlock;      // always 0
    ULONG   udb_Checksum;       // checksum of this block

    ULONG   udb_HashTable[];    // variable number of hash table entries

    ULONG   udb_Reserved;       // always 0
    ULONG   udb_OwnerXID;       // owner UID/GID
    ULONG   udb_Protect;        // protection bits
    ULONG   udb_Junk;           // not used (always 0)
    char    udb_Comment[92];    // comment as a BCPL string
    ULONG   udb_Days;           // creation date and time
    ULONG   udb_Mins;
    ULONG   udb_Ticks;
    char    udb_DirName[36];    // name as a BCPL string
    ULONG   udb_Link;           // pointer (key number) to object linked to
    ULONG   udb_BackLink;       // back pointer to previous hard link header
    ULONG   udb_HashChain;      // next entry with same hash value
};
```

```

        ULONG   udb_Parent;           // pointer back to parent
        ULONG   udb_DirList;          // DOS\4 and DOS\5 use this for the dir cache
        ULONG   udb_SecType;          // ST.USERDIR
};

```

A new block type added in support of DOS4 and DOS5 is the file list block. This is where directory information is cached by the file system. When looking at a file list block, is it critical to check the fhl_Type field of the block to ensure that it is a known block format. Unknown block formats should be ignored. On a DOS4 and DOS5 partitions, the file system is able to rebuild cache blocks if it finds older versions present, or if it finds no cache block for a given directory.

```

struct FileListBlock
{
    ULONG   fhl_Type;           // T.DIRLIST
    ULONG   fhl_OwnKey;          // key of this block
    ULONG   fhl_Parent;          // key of parent directory
    ULONG   fhl_NumEntries;      // amount of data on this block
    ULONG   fhl_NextBlock;       // next block in sequence of file list blocks
    ULONG   fhl_Checksum;        // checksum of this block
};

```

Following the above structure on a disk block comes the actual cache information for the directory. There is one cache entry for every file and directory within the cache. These entries vary in size based on the length of the file name and comment for the entry. A cache entry looks like:

```

struct ListEntry
{
    ULONG   fle_Key;             // Key (fhh block) of file
    ULONG   fle_Size;            // size in bytes
    ULONG   fle_Protection;      // protection bits
    ULONG   fle_OwnerXID;        // owner info
    UWORD   fle_Days;            // date (allows 179 years)
    UWORD   fle_Min;             // 0..(60*24 - 1)
    UWORD   fle_Tick;            // 0..3599
    UBYTE   fle_Type;            // ST.XXXXX (all fit in a byte)
    char    fle_FileName[];      // variable length BCPL string
    char    fle_Comment[];       // variable length BCPL string
};

```

These are the values for the various constants referred to above:

```

// block type constants
#define T.DELETED 1
#define T.SHORT 2
#define T.LONG 4
#define T.DATA 8
#define T.LIST 16

DCFS_REV 1
DIRLIST_KEY 32
DIRLIST_MASK 0xffffffff
T.DIRLIST (DIRLIST_KEY | DCFS_REV)

ST.FILE -3
ST.ROOT 1
ST.USERDIR 2

```

DOS Workarounds

V37 dos.library has a few important bugs. These bugs are fixed in V39, and it is easy to work around many of these bugs under V37 to make sure you don't get into trouble.

AttemptLockDosList(). Under V37, AttemptLockDosList() would sometimes return 1 on failure instead of NULL. To check for failure of AttemptLockDosList(), simply use:

```
dl = AttemptLockDosList();
if (!dl || (dl == (struct DosList *)1))
    /* failed */
```

FGets(). Under V37, FGets() will overrun the buffer you give it by one byte, if it doesn't encounter a line feed or EOF before filling up your buffer. The work around is very simple: pass buffersize-1 to FGets().

ParsePatternNoCase(). Until V39, ParsePatternNoCase() did not correctly treat character classes as case-insensitive. That is [a-z] was different than [A-Z]. The work around for this bug is to simply convert all characters of the pattern to upper case before passing it to ParsePatternNoCase(). Just use the utility.library/ToUpper() function to convert all characters of the pattern to upper case.

SetVBuf(). Until V39, the SetVBuf() function didn't do anything. Starting with V39, it actually does in fact set the buffer sizes. If you use any of the buffered DOS IO calls, you can get a noticeable increase in performance by increasing the buffer size of your files. A good size is 4K. You can simply add a SetVBuf() call after opening each of your files. It won't do anything under V37, and will work under V39. In most cases, it is not even necessary to check for failure of SetVBuf(). When the call fails, it just leaves the file handle with the same buffering it had previously. That's fine, you just won't get the performance increase for that run of the program.

SetVBuf() was not activated in the initial release of 3.0, it only becomes available with 3.01.

Commodities Library

The commodities.library underwent a major overhaul in V38. The result is a much smaller, much less CPU hungry, and much less memory hungry library. Many bugs were also found and fixed. Finally, a few new features were included. All of the commodity programs shipped with the system were also revised.

ReadArgs(). Commodities now use ReadArgs() instead of the custom command-line parsing routines in amiga.lib. This makes them more consistent with the rest of the system. We encourage third party commodity writers to also switch to ReadArgs().

Keyword Synonyms. ParseIX() now understands many synonyms for keys and qualifiers. The intent was to provide more consistent naming, and to decrease the likelihood the user would make a mistake while entering a key sequence. For a complete list of all the keywords and synonyms refer to the *V39 Release Notes* available from Commodore's CATS department (part number: V3.0)

Input Events. A bug that was found in a few applications during V39 development was the incorrect initialization of InputEvents inserted into the input.device's input stream. These events can work most of the time, and suddenly not work with specific applications. Of specific interest, the inputEvent->ie_SubClass was found to often contain garbage. This field should generally be set to 0. Another field that is often incorrectly initialized is the ie_TimeStamp field. Make sure this field has something meaningful in it. Note that using the IND_WRITEEVENT input.device command automatically initializes the ie_TimeStamp field.

Bugs In commodities.library. Under V37, input events output by translator objects were inserted into the input stream after the commodities.library input handler. This was contrary to the documentation. Starting with V38, events generated are inserted immediately following the translator object. This means commodity objects coming after the translator now get to see the translated events, while in V37, they wouldn't.

Another bug with translator objects is that they insert their attached list of input events in reverse order. This cannot be fixed due to compatibility problems. It is easily worked around by arranging the input events in the reverse order of how you want them to be inserted. The InvertString() function in amiga.lib generates an inverted list of input events, which is just fine for translator objects.

Starting with V39, input events generated by translator objects are stamped with the same time as the input event that activated the translator. This helps applications that look at the time of input events.

In V38, the ParseIX() function has a bug which prevents the following keys on the numeric keypad from being used in commodities: . 9 () / * +. This bug is fixed in V39.

Until V39, input events of type IECLASS_NEWPOINTERPOS never got their extended tag list data copied when it needed to be. This can cause general confusion or crashes if a sender object is used on an event of that type. The only solution is to refrain from sending messages of type IECLASS_NEWPOINTER.

Until V39, the InvertKeyMap() function never initialized the ie_SubClass and ie_TimeStamp fields of the input event it created. The workaround to this bug is simple:

```
inputEvent->ie_SubClass      = 0;
inputEvent->ie_TimeStamp.tv_secs = 0;
inputEvent->ie_TimeStamp.tv_micro = 0;
if (InvertKeyMap(ansiCode, inputEvent, NULL)) ...
```

Mount Lists

The Mount procedure got a major overhaul in V38. The major change from the user's perspective is that the handlers that get mounted in the system can now be controlled exclusively through Workbench by dragging icons around. This is especially important in light of CrossDOS.

New File Format. Although the traditional DEVS:MountList file is still fully supported, a new method of storing mount information is now recommended. The new format involves simply splitting up the various entries that used to be in a mountlist, into separate files. Each file controls a unique handler. The format for the information in the mount files is the same as what is used in a traditional MountList with three exceptions. For example:

```
/* Aux-Handler mount entry */
Handler   = L:Aux-Handler
Stacksize = 1000
Priority   = 5
```

The differences with old style MountLists are as follows:

- ☐ Only a single device can be defined per file.
- ☐ The name of the device is not specified in the file and is instead the same as the name of the file.
- ☐ The # at the end of the entry can now be omitted.

Every parameter that can be specified in a mount file can also be specified in the tooltypes for the icon of the mount file. The parameters in the tooltypes override any equivalent parameter setting present in the mount file. The format for the tooltypes is straightforward. For example:

```
LOWCYL=0
HIGHCYL=79
```

This lets the Workbench user easily control certain attributes associated with any given mount entry. For example, the WB user can easily set the size of RAD from tooltypes.

If the user wishes to only use your handler once in awhile, he can drag the icon for the handler in the Storage drawer. This prevents the handler from being mounted on every boot. If the user wants to use the handler for a given session, he can then simply double-click on the mount entry icon to get it mounted.

It is fairly simple to create an installation procedure that deals with both old style and new style mount files:

```
if (V38)
{
    /* Create a mount entry file. If the user indicates he wants to use the
```

```

    * handler permanently, put it into DEVS:DOSDrivers. Otherwise, just put
    * it into SYS:Storage/DOSDrivers. Don't forget to provide an icon for
    * the mount entry.
    *
    * If the mount entry is in DEVS:DOSDrivers, it will get mounted
    * automatically when the system is rebooted. The user can also mount
    * it for the current session by double-clicking on it.
    */
}
else /* V37 or before */
{
    /* Create an old-style mount entry and append it to DEVS:MountList.
    * Add a Mount statement in the user-startup of the system.
    */
}

```

File System Resource List. Mount now has the ability to fish out file systems from the FileSystem.resource list. When mounting a file system, Mount first look in the FileSystem.resource to see if the system already contains a file system of the correct DOS type. If there is such a file system, its seglist is used for the new handler.

For example, the CrossDOS file system registers itself in the FileSystem.resource the first time it is used. Any subsequent attempts to mount a handler having the same DOS type as the CrossDOS file system results in that handler using the same seglist as the original CrossDOS handler.

If a file system seglist is reentrant, it should be added to the FileSystem.resource list of file systems, which will enable it to be used in the manner described above. Be sure the seglist is totally reentrant before doing this (no global variables). Consult the Rom Kernal manual to learn how to add a file system to the FileSystem.resource list.

The new FORCELOAD keyword can be specified in mount entries. When its value is set to 1, it tells mount to always load the file system for this handler from disk, even if it is present in the FileSystem.resource list. This can be quite handy during the testing phase of a handler.

Unit Keyword. Stream handlers have always had the advantage of being able to let the user provide startup arguments to the handler via the Startup keyword. For example:

```
Startup = "this is a bunch of options"
```

The value of the parameter is available to the handler as a BSTR in the dol_Startup field of the handler DosList structure. This only works for stream handlers though, file system handlers make use of the dol_Startup field as a BPTR to the FileSysStartupMsg created for them by the Mount command.

The Unit keyword can now be used to passed textual options to a file system handler. If a string is specified in the mount entry, then the fssm_Unit field of the file system's FileSysStartupMsg will be a C pointer to the NULL-terminated argument string, instead of being the traditional unit number.

New Values For GlobVec. The GlobVec keyword in a mount list can now be set to -2 and -3 in addition to the other values already supported. Here is a table of possible values:

GlobVec	
Value	Meaning
0	Default global vector, used for handlers written in BCPL. Startup packet comes in a register.
-1	No global vector, used for C and ASM handlers. Process has to wait for startup packet.
-2	Same as -3, except for handlers written in BCPL.
-3	No global vector, used for C and ASM handlers. The difference with -1 is that the handler can do DOS IO prior to returning its startup packet. On the flip side, the handler is responsible for making sure there is only one instance of itself loaded by providing correct arbitration around the poking of the dol_Taskfield of the DosList structure. With a GlobVec of -1, this is taken care of by DOS.

Even though the values -2 and -3 are only allowed starting with the V39 Mount command, the V37 dos.library handles those values correctly. It's just that prior to V37, the Mount command would refuse to process these values.

Version Command

There has been some confusion about what the format of version strings are. The exact format is detailed in the *Amiga User Interface Style Guide* on page 110. The format is:

\$VER: <name> <version>.<revision> (dd.mm.yy)

The string starts with \$VER: followed by a space, followed by <name>. <name> is the name of the program. <name> can NOT contain any spaces. You can use underscores to achieve a similar effect.

After <name> comes a single space, followed by <version>. <version> is the major version number for the program. There should be no leading zeros.

After <version> comes a single space, followed by <revision>. <revision> is the minor version number for the program. There should be no leading zeros.

Following the revision number comes a space, and then the date. The date is specified in numeric form only, with no leading zeros on any of the components. First comes the day of the month, then the month, then the two digits year. In the future, a four digit year format will also be supported, but not yet.

Embedding a version string in the exact format described above will let the C/Version command find the version string. In the future, there will be a system call to enable applications, and other system tools, to obtain the version information from files easily.

ROM Tags. The version command not only looks for the \$VER: version strings, it also search executable files for standard ROM tags. ROM tags only contain a version number, they have no revision or date fields. The recommended approach is to store a pointer to a version string in the RT_IDSTRING field of the ROM tag. This string should be in the same format as the normal version string, except without the "\$VER: " prefix.

An executable file containing a ROM tag with a properly initialized RT_IDSTRING field does not need any other version information in it, such as a separate \$VER-style string.

Leading Zeros. Leading zeros are not supported as part of the version and revision numbers. So "2.04" is not a valid version/revision pair as far as the version command is concerned. The reason this is so is because of the VERSION and REVISION command-line options of the Version command. These allow the version and revision of a file to be checked by a script such as:

```
Version asl.library VERSION 38 REVISION 4
```

The above command will return 0 if asl.library is greater or equal to 38.4, and 5 if it is not. Specifying:

```
Version asl.library VERSION 38 REVISION 04
```

Does the same thing. This means that from the standpoint of the Version command, 2.04 is the same as 2.4, and thus 2.04 evaluates as being greater than 2.1, which is not the desired effect. So, the version and revision numbers must not be treated as a floating-point number, but as two separate and distinct integers.

This brings up the concept of user versions and internal versions. V37 is known to the public as Release 2.04, and V39 is known as 3.0. All the version strings in the system software use version strings starting with V39. The recommended approach is as follows:

- ☐ Assign your products user-space numbers of the form 2.04 and the like.
- ☐ Assign the various components of your distribution the same version numbers as for your product. For example, "2".
- ☐ Assign each individual components of your distribution their own unique, monotonically increasing revision numbers. These numbers have nothing to do with the user-space number of your product as a whole.

An example is in order. Assume a word processor called "SliceAndDiceWord", which is at version 4.03. All the files composing the distribution would have a version number of "4", and a file-specific revision number:

Package number : SliceAndDiceWord 4.03
Main executable: SADW 4.1423
Support library: SADW.library 4.4129
README file : SADW.README 4.6

When SliceAndDiceWord 5.0 comes out, all the version numbers of the files included with the distribution get bumped to 5, and the revision numbers start at 0 again.

Miscellaneous

This section presents miscellaneous tidbits of information worth noting when developing applications.

Overscan Names. Starting with V38, the two user-settable overscan regions are known as "Text Size" and "Graphics Size". Please use the following names in your user documentation:

OSCAN_TEXT --> "Text Size"
OSCAN_STANDARD --> "Graphics Size"
OSCAN_MAXIMUM --> "Extreme Size"
OSCAN_VIDEO --> "Maximum Size"

Audio Beep. Starting with V38, the system provides standard support for audio beeps, and complete sampled sounds, as a replacement for DisplayBeep(). You may wish to rely exclusively on this feature instead of providing your own application-specific control over an equivalent feature.

ToolType Comments. We strongly encourage you to follow the convention adopted in V38 with respect to specifying all tooltypes supported by an application within the icon. Any tooltypes not in use can be commented out by putting a set of () around the tooltype. This causes Workbench to ignore the tooltype, yet still leaves it there for the user to see and uncomment.

Keymap selection. If you wish to offer application-specific control over the keymap being used, you have to load the keymaps you wish to use from disk yourself. Some applications were using the SetMap command to load keymaps into memory for them. SetMap is no longer part of the system, so this technique will no longer work. The DevCon disks contain an example showing how to load keymaps from disk.

Note that if you wish to offer a list of keymaps to the user, you should be looking in the KEYMAPS: assign list for the available keymaps. To work under V37, If this assign list doesn't exist, you can look in DEVS:Keymaps. The example code shows how this can be done.

Appendix A: V39 Disk Changes

Following is a description of most of the important changes made to the disk-based system software between V39 and the initial release of V39. This doesn't cover changes made to modules discussed in other talks, such as the addition of datatypes and AmigaGuide.

C/Install

- o Doing "INSTALL DF0: CHECK" on a disk formatted with dir cache now reports the disk as either "DC-OKs" or "DC-EFs" as appropriate.

C/TPrefs

- o Requires Kickstart V39 and handles the new palette, pointer, and wallpaper preferences.
- o Is smarter about when it needs to reset the WB screen. Now sends an initial IP SCREENMODE message to Intuition to force the initial Workbench screen to be interleaved even if there is no screenmode.prefs file.
- o Now uses utility.library 32-bit math routines and uses datatypes.library to load sound samples instead of custom @SVX code.
- o Causes much less screen flashing when changing fonts. No longer has the QUIR/5 command-line option. (Removing this option allows a few K of RAM to be saved in every machine. Gives better error messages when it can't find a font or a picture.

C/Mount

- o Added support for GlobVec of -2 and -3.
- o No longer considers empty strings an error. This allows something like:

```
Device = ""
```

to work. This fix will be helpful to the Envoy software. Note that this bug is in V38, so there will soon be many many Mount commands out there that don't support empty strings.

C/RequestChoice

This is a new C: program that lets AmigaDOS and ARExx scripts use the Intuition EasyRequest() feature. The template is:

```
TITLE/A,BODY/A,GADGETS/M,PUBSCREEN/K
```

TITLE

Specifies the title of the requester.

BODY

Specifies the text of the requester. Linefeeds can be embedded using *n.

GADGETS

Specifies the labels for the different gadgets at the bottom of the requester.

PUBSCREEN

Lets you specify the name of a public screen to open the requester on.

The number of the selected gadget is printed at the console on exit, and is returned as the secondary return value (seen as the Result2 variable in the Shell).

C/RequestFile

This is a new C: program that lets AmigaDOS and ARExx scripts use the ASL file requester. The template is:

```
DRAWER,FILE/K,PATTERN/K,TITLE/K,POSITIVE/K,NEGATIVE/K,ACCEPTPATTERN/K,REJECTPATTERN/K,SAVEMODE/S,MULTISELECT/S,DRAWERONLY/S,NOICONS/S,PUBSCREEN/K
```

DRAWER

Specifies the initial contents of the Drawer gadget.

FILE

Specifies the initial contents of the File gadget.

PATTERN

Standard AmigaDOS pattern. Specifies the initial contents of the Pattern gadget. Not providing this option causes the file requester not to have any Pattern gadget.

TITLE

Specifies the title of the file requester.

POSITIVE

Specifies the text to appear in the positive (left) choice in the file requester.

NEGATIVE

Specifies the text to appear in the negative (right) choice in the file requester.

ACCEPTPATTERN

Specifies a standard AmigaDOS pattern. Only files matching this pattern will be displayed in the file requester.

REJECTPATTERN

Specifies a standard AmigaDOS pattern. Files matching this pattern will not be displayed in the file requester.

SAVEMODE

Specifying this option causes the requester to operate in save mode.

MULTISELECT

Specifying this option causes the requester to allow multiple files to be selected at once.

DRAWERONLY

Specifying this option causes the requester to not have a File gadget. This effectively turns the file requester into a directory requester.

NOICONS

Specifying this option causes the requester never to display icons (.info) files.

PUBSCREEN

Lets you provide the name of a public screen to open the file requester on.

The selected files are printed on the console at exit, enclosed in double quotes and separated with spaces. The command generates a return code of 0 if the user selected a file, or 5 if the user cancelled the requester.

C/SetFont

o Now sends ESC-C to the console handler only after doing SetFont() in the console window's rasport. The only reason SetFont ever worked was by luck, since the escape sequences were buffered by DOS and flushed only after the program exited.

o No longer crashes when passed a font name with more than 80 characters. Now returns with ERROR_BAD_NUMBER if asked for a font < 4 points. No longer opens console.device.

C/SetPatch

- o Activates AA chips.
- o On some systems, NMI interrupts could be caused due to hardware bugs. These NMIs can cause software/hardware to perform incorrectly and has caused major performance problems in some systems. In the V37 ROM, the NMI vector pointed at a generalized routine that saved all registers, does some work which ends up doing nothing, restores all registers, and exits. This patch will, under V37 ROMs, move that vector to point directly at an RTE.
- o SetPatch no longer will read the longword after the end of each of its hunks.
- o Now checks for the CPU type before trying to open the 68040.library. This is so that the library does not even get loaded if you don't need it. (The 68040.library also does the check.)
- o Does not turn on the caches if the instruction cache is not already on. This is such that the bootmenu option will not be "undone" by SetPatch.

C/Version

- o Now does versions of gadget and datatypes classes.
- o Significant change in behavior required when not using the standard version string format:
Junk following the date parameter is printed whenever the FULL option is provided on the command line. Incorrectly formatted components of a version string are printed whenever the FULL option is provided.
Note that as always, specifying incorrectly formatted version strings will cause the VERSION and REVISION command line options of the Version command not to work as expected. Basically, using these keywords, will result in output like "2.1 < 2.04", which is not what is usually expected.
The incorrect version string formats are not supported by the Installer. It will behave much like the VERSION and REVISION cmd-line options of the Version command.

Classes/Gadgets/colorwheel.gadget

The colorwheel.gadget is a new BOOPSI class to create color wheels to allow color selections by the user. See colorwheel.doc for more info.

Classes/Gadgets/gradientallder.gadget

The gradientallder.gadget is a new BOOPSI class to create sliders having a gradient background. This is to be used in combination with the colorwheel.gadget. See gradientallder.doc for more info.

Devs/Datatypes

This new directory in V39 is the place where a user puts datatypes.library descriptor files.

Devs/printer.device

- o Added small amount of code to support AA 8-bit HAM mode printing.
- o Also now uses GetBitmapAttr() if running under V39 to obtain dimensions of source Bitmap structure.

L/FastFileSystem

- o Equivalent to the V39 ROM file system, including DOSV5 support. See ROM release notes for more info.

Libs/68040.library

This is a support library for 68040 systems. It addresses a number of issues with that CPU. There are no publically callable functions in this library.

Libs/amigaguide.library

The amigaguide.library is a new system module providing a means whereby developers can easily add on-line help to their applications. See amigaguide.doc for more info.

The amigaguide.library was previously available for use in commercial applications. This is the first release of the library as a part of the core operating system. Changes since the previous release include:

- o All new V39-only version uses datatypes.library for object handling.
- o Now has an AppWindow, plus an "Open..." menu item. To open a new document, AmigaGuide performs a LINK to it, so that you can RETRACE back through the documents that you have opened.
- o Now supports word wrapping and proportional fonts.
- o Now supports font attributes.
- o Now supports embedded objects. This means that you can LINK to any type of object that the user has a DataTypes descriptor and class for---such as pictures and animations.
- o Modified how files are located, to search for a localized help file. Now searches in the following order. For each preferred language

```
PROGDIR:Help/<language>/<name>
HELP:<language>/<name>

PROGDIR:<name>
<name>
```

This matches how locale.library searches for catalog files. For compatibility, each directory from the AmigaGuide/Path environment variable are also searched.

- o Renamed S:help.guide to HELP:<language>/Sys/amigaguide.guide
- o All sub-databases are now opened localized.

Libs/asl.library

- o Added support for the new WA:HelpGroupWindow Intuition tag. ASL requesters now inherit the group id of their parent window.

- o Removed V37 code. The library now requires V39.

- o Added support for OSCAN_VIDEO in the screen mode requester.

- o Fixed rendering bug with selected items in the file requester. Depending on the size of the requester, sometimes the rendering didn't occur correctly. Fixed font requester layout bug with checkmark labels using certain fonts.

- o Now monitors the size numeric gadget more closely in the font requester.

- o ASL now uses scaled checkmark imagery in the font and screenmode requesters. Also now uses ROM busy pointer instead of custom one.

Libs/commodities.library

- o Fixed ParseIX() bug introduced in V38 preventing the following keys on the numeric keypad from being used in commodities: 9 () / * + .

Libs/datatypes.library

datatypes.library is a new system module providing transparent data handling abilities to applications. Application developers can register their data format with datatypes.library and provide a class library for handling their data within other applications. See datatypes.doc for more information.

Libs/diskfont.library

o Marks DUPLICATE fonts using one of many free bits in the TextFont Extension. These are fonts opened by diskfont, but match some other font already on the public font list by name, y size, relevant style, and relevant flags. The font is only marked as a duplicate if a NON-DUPLICATE occurrence of this font already exists.

DUPLICATE fonts are ignored by OpenFont() if the caller is not tag aware even if there is a font on the list which has a better implied DPI.

o Tightened up requirements for source font when scaling. Specifically, an already scaled RM font of an exact y size match but differing by DPI will not be used as the source for a bitmap scaled font (adding scale error, to scale error).

o The code which generates the outline fonts now does comparison for OT_Dotsize when it calls OpenFont(); essentially honoring the caller's request for an OT_Dotsize other than the default that diskfont will provide when creating outlined fonts. This makes it possible to have multiple outline fonts which differ only by OT_Dotsize.

o Underlined outline fonts are now supported if the engine can manage it when rendering the glyphs. New tags in diskfonttag.h[] were added for this purpose. V39 diskfont will ask for underlining if requested in the TextAttr, or TextAttr, but if not supported by the engine, then it will simply try to open, or create a non-underlined version. So there is no change in behavior for bullet fonts in that it already opens a non-underlined version. The difference is by asking OpenFont() for a non-underlined version. It does not decide to create another copy because of the result returned from WeightMatch().

Of interest, our own Text() function when used with SetSoftStyle() does a broken underline (just one pixel, but its definitely not solid), so diskfont.library V39 first asks for broken underlining from the engine, and then solid. If neither form of underlining is supported, then the above is true.

o Tag for algorithmic strike-through added to diskfonttag.h[]; this is another feature supported in Final Copy, and could be requested if wrapped in a bullet interface.

o Underlining off, double solid, and double broken underlining also defined for the new OT_Underlined tag.

Underlining is not an intrinsic style for outline fonts. This does not actually absolutely have to be the case, but we lack a tag to indicate intrinsic underlined style now, so this is not a new behavior, or limitation. The assumption being that underlining is one of those things that is probably best done algorithmically by the application, or for the purpose of diskfont, by the engine. Therefore this also leaves open the possibility of placing the underlining code within diskfont for engines that do not support it should we wish to do so in the future.

Libs/iffparse.library

o Fixed three bugs in the OpenClipboard() function. First bug was that if the clipboard.device couldn't open, two calls to FreeSignal() were made with uninitialized values as parameters. The result of this was a corrupt signal mask in the Task field. Second bug what the OpenDevice() was called with an IO request that didn't have a valid MagPort pointer

in it. Finally, the two message ports allocated by the function were not being initialized correctly and would cause a system crash if a message ever got to either of them.

Prefs/##

o Now restore task->tc_UserData on exit of prefs editor.

Prefs/Font

o Default for WB icon text mode when there is no .prefs file is now JAM2 instead of JMI1. This matches the ROM.

Prefs/IControl

o Added Mode Promotion gadget when running on AA machines

Prefs/Input

o "Show Double-Click" now operates asynchronously. That is, while the double-click sample is displayed, it is still possible to click on other gadgets.

Prefs/OverScan

o Now bounds check values read from prefs file against what the gfx database says. This avoids problems with V38 overscan prefs files, and with running VGAOnly or not.

Prefs/Palette

o Brand new interface using the colorwheel and gradientalider gadget classes. Also allows pen spec editing.

Prefs/Pointer

o Now opens on the Workbench screen if it can obtain three exclusive pens. Allows editing of the Normal and Busy pointers and supports Low-Res and High-Res pointer resolutions.

o Uses datatypes.library for clipboard support.

o Semaphore around save during Test. This is to prevent confusing error requesters if the user presses the Test button several times while waiting for the pointer to appear.

o Added "Load Image..." menu item in the "Edit" menu.

o Uses (SA_LikeWorkbench, TRUE) when using a custom screen.

o Increased width of window by 20 pixels to make room for longer strings.

o Added MOREMAP tooltype and command-line option. This causes paste and import image to not remap.

Prefs/screenMode

o Now skips the default monitor when scanning the mode list instead of skipping PAL or NTSC. This means that the prefs file will never contain "default monitor" as a mode, it will always contain an explicit mode.

Prefs/sound

o Now uses datatypes.library for sound loading and playing. Also uses a Datatypes filter in the load sample requester so that only sounds will be shown.

Prefs/WBPattern

- o Added support for picture backdrops.
- o Added support for Workbench screen backdrop.
- o Defined a more efficient chunk for the preference data.
- o Fully supports the First4/Last4 Workbench color scheme.
- o Uses datatypes.library for clipboard support.
- o All internal bitmaps now max out at a depth of 3 (less memory usage).
- o Rearranged the gadgets.
- o Semaphore around save during Test. This is to prevent confusing error requesters if the user presses the Test button several times while waiting for the pattern to appear on the Workbench.
- o Added "Load Image..." menu item in the "Edit" menu.
- o Added MOREMAP tooltype and command-line option. This causes paste and import image to not remap.

S/Startup-Sequence

- o Removed "Echo" statement. That means the initial shell window will no longer be displayed on bootup on a standard system. This means a noticeable decrease in bootup time on slower machines (it takes about a second to open and close the shell window). It means even more savings in time when the WB has more bitplanes. Finally, it also looks a heck of a lot more professional.
 - o Added Ci in front of every pertinent command. This speeds up the S-S noticeably on floppy systems, since normally the current directory is scanned first for a command, and then Ci. By prefixing Ci, it no longer checks the current directory making things a whole lot faster.
 - o Added deferred assign for HELP:.
 - o Added multi-assign of LIBS: to SYS:Classes in support of the new disk-based BOOPSI classes.
 - o Now runs VGMOnly monitor file if it is installed.
- #### Storage/Monitors/##
- o Added DbINTSC and DbIPAL in support of AA scan doubling.
 - o Added VGMOnly monitor.
 - o Added AA support throughout.
- #### System/DiskCopy
- o Enabled code to correctly report read errors. Under V38, read errors are reported as write-errors.
- #### System/Format
- o Requires V39.
 - o Added directory caching support in the form of an extra gadget in the main Format window, and some new command-line options. Command-line template is now:

DEVICE-DRIVE/K/A,NAME/K/A,OPS/S,FES/S,INTL-INTERNATIONAL/S,
NOINTL-NOINTERNATIONAL/S,DIRECTORY/S,NOIRCACHE/S,NOICONS/S,QUICK/S

- o Uses GTLV_MakeVisible tag in its device listview.
- o Fixed bug where long volume names would run off the right edge of the main window by using the new GTTX_Clippped gadtools tag.

System/Intellifont

- o Renamed from Fountain to Intellifont.

- o Bare minimum needed to fix visual bug evident when running under V39 gadtools; no longer uses GTLV_Selected tag, though still does not use V39 style selection via GTLV_ShowSelected tag.

Tools/Commodities/MouseBlanker

This is a mouse blanking commodity. It blanks the mouse when you start typing and unblanks it when the mouse is moved or any of the mouse buttons are hit.

Tools/HDTToolBox

- o Brand new GadTools interface.
- o Now determines the version number of a file system automatically instead of requiring it to be entered manually.
- o Fixed a great number of bugs.

- o When CD-ROM drive was hooked on the SCSI bus and the disk was inserted, HDTToolBox hanged up because the program assumed only 512 bytes blocks. It now checks the size with the "READ CAPACITY" command, and if it fails, sends "INQUIRY" command and gets device type. If it's a CD-ROM device, sets block size to 2048.

Tools/IconEdit

- o Implements the First4/Last4 Workbench color scheme.
- o Uses datatypes.library for clipboard support and picture loading/saving. Therefore, you can now load any picture type that you have a class for. When loading or pasting a picture, the colors are mapped to the Workbench GUI colors (either 2, 4 or 8 colors). 8 color icons will work with any depth Workbench screen.
- o Default clipboard viewer is now "Multiview CLIPBOARD".
- o Checks to see if icon already exists when saving a picture or source file and won't overwrite it.
- o File requester's initial position now matches the preference editors.
- o Now draws the coordinates in the window title bar using the pen spec.
- o Was reloading the icon image after save, even if save failed.
- o Will now properly undo after loading a new icon image.
- o Added MOREMAP tooltype. This causes paste and import image to not remap.
- o Fixes Enforcer hit caused when dropping Drawer icon (without a drawer) into the AppWindow.

Tools/ShowConfig

- o Added AA chipset support.

Utilities/MultiView

This is a generic view-everything utility that uses datatypes.library for its object handling.

3.0 Installation Procedure

- o It is no longer necessary to boot from the Install disk.
 - o Install Languages no longer exists. This is now merged into the main installation script.
 - o Install Printers no longer exists. The user can easily drag a printer driver icon from the Storage disk to DEV5:Printers
 - o The disk now includes the 68040.library, which gets copied to the HD when performing an install
 - o The various Prod_prep scripts were enhanced and work better.
 - o There are many little utilities in the C: directory which aid the main install script to do a better job.
- Changes to the install script include:
- o Requires V39 ROMs
 - o Now integrates the old "Install Languages" script. When starting the main install script, you are now asked whether to perform a complete installation, or only update the languages.
 - o It now virtually always makes the right decision as to where the WB files should be copied. As a result, if the user picks Move mode, he will not be asked where the files should go.
 - o Now preserves the tooltypes of many icons. This will avoid zapping the stuff put there by users, making the update process much more transparent.
 - o Now preserves the locations of most files. For example, if the user moved Blanka in WBStartup, it will be correctly updated in WBStartup, and left there. In V38, the copy in WBStartup would be deleted and the new version installed in Tools/Commodities.
 - o Now preserves left out icons. The UpdateWBFiles program ensures that the contents of the .backdrop file are up to date which removes the need to delete the file.
 - o Now updates V37 font prefs files to V38/V39 format.
 - o Now deals with the "storage3.0" disk
 - o Now asks which keymaps to install.
 - o Installs the 68040.library.
 - o Knows about systems with MapROM, and installs a special startup-sequence.
 - o On an NTSC system, always copies the NTSC monitor to DEV5:Monitors, and copies PAL to DEV5:Monitors on PAL systems.
 - o When installing languages, deletes languages the user did not choose. This is to remove old V38 files that might be hanging around.
 - o Does better sniffing to determine whether an A2090 is present, and displays extra info to inform the owner that some files need to be copied.
 - o Will optionally reboot the system for the user at the end of the installation procedure (after asking permission first, of course).
 - o Asks all its questions up front. Once the questions are answered, it starts the actual installation.

- o Now deletes all C= printer drivers and keymaps before starting the installation process. This was previously done right before re-installing them. This is to increase the likelihood that things will fit on the HD by deleting obsolete stuff ahead of time.
 - o Does slightly better icon positioning to avoid things moving "by themselves".
 - o Now copies the A2232 port-handler and A2232 aux-handler when needed.
- Changes to the HDSetup script include:
- o Will now partition the HD and format it automatically instead of requiring a reboot.
 - o The partitions created are now called Workbench (HD0) and Work (HD1). HD0: is created as 8M.
 - o DOS\3 is now used by default, and the DMA mask is set to 0xffffffff instead of 0xffffffff
 - o The version number for the file system being installed on a machine is now extracted from the file system load file instead of being hardcoded in the prep scripts
 - o After completing the reselection on, reselection off, or update superkickstart option, the system is rebooted automatically (after asking the user of course).
 - o Updating an A3000 super kickstart no longer requires the user to hit RETURN. Errors are also detected now.

Appendix B: Release 3.01 Disk Changes

Following is a description of most of the important changes made to the disk-based system software between Release 3 and Release 3.01. This doesn't cover changes made to modules not discussed in this talk. It also doesn't cover any changes after December 1992. Since 3.01 is still under development as of this writing, more changes are likely to be made to the software prior to release.

C/ConClip

- o Now consumes over 3100 bytes less RAM when running. That means over 3K more RAM in every system we ship.
- o Localized
- o Now keeps `iffparse.library` and `clipboard.device` closed whenever they are not in use.

- o Added `CLIPUNIT` synonym to existing `UNIT` command-line option. This is to make it comply with the style guide and be more consistent with other programs such as `MBPattern`. The template is now:

`CLIPUNIT=UNIT/N,OFF/S`

C/Copy

- o Now correctly handles failure of `SetProtection()`, `SetComment()`, and `SetFileDate()`. Because `SetComment()` fails on NFS and on CrossDOS, `SetFileDate()` never was done on files, making the `CLONE` option not fully work. All three functions are considered failures only if `IOErr()` reports something different than `ERROR_ACTION_NOT_KNOWN`.
- o In case of error after a call to `DupLock()`, an error code was being set always to `ERROR_NO_FREE_STORE` instead of using the result of `IOErr()`.
- o The default size for the buffers used was always equivalent to `(BUF-0)` which caused the buffers to be the size of the files being copied. This was contrary to the docs, and caused problems when copying large files through Envoy, as it could easily eat up all the memory in the system, not leaving enough for the memory needed by Envoy. The default size is now `BUF-128` which gives a 64K buffer.

C/IPrefa

- o Added needed support for the new display position control now offered by `OverScanPrefa`.
- o If there is no `icontrol.prefs` file, `IPrefa` sets the default for Mode promotion to `ON`. That means machines will have mode promotion on by default.
- o Now ensures that requested width, height, and depth for `Workbench`, fall within the allowed range as defined in the `graphics.database`.
- o Only change is the version number. Versions 39.1 and 39.2 of this program were accidentally numbered 38.1 and 38.2.

C/Protect

- o Fixed error reporting. When used with wildcards, it would generate errors of the form "Can't set protection for #?" instead of giving a descriptive file name, and cause of error.

C/SetPatch

- o Due to a bug fix late in the game, the filesystem broke with `ExAll()` on DCFS file systems if the lock passed in was not `Examine()`ed first. This patch fixes this by forcing an `Examine()` on a lock before `ExAll()` is called. It only does this on systems with V39.22 `dos.library` since only that ROM has the problem with DCFS `ExAll()`.
- o `ChangeVBitMap()` was not properly doing the bitplane swizzle needed for 8-bit HAM mode. The reason that this didn't show up earlier is that in the relationship between the bitplane pointers was the same in the original and new bitmaps, the bug would correct itself. So, you would only see this bug if your memory was fragmented, or if some other allocations got in between the separate allocations of the bitplanes.
- o `ScrollVPort()` had the same bug as `ChangeVBitMap()`.
- o The patch locks the `ActivViewCPrsemaphore`, swizzles the bitmap, calls the old entry point, and unswizzles the bitmap, when called on a HAM-8 viewport.
- o The patches for `graphics.library/ScrollVPort()` and `ChangeVBitMap()` now bump the graphics revision number to 90 if it is 89. This is so that installing the setpatch on an A1200 or A4000 will not break people using work-arounds for the bug.
- o When calling `BitBitMap()` with both source and destination interleaved, and a mask of -1, the low byte of d7 would be changed to the bitmap depth on exit. Patch saves d7, calls old, and restores it.
- o Added the graphics monitor/view association hash patch.
- o `BitMaskBitMapRastPort()` used to interpret the mask data incorrectly if both the source and destination bitmaps were interleaved.
- o Now includes the CP2024 Conner patch from the V37 SetPatch.

C/Version

- o Obtaining the version of printer.device while it is loaded in memory would trash memory. This is because printer.device doesn't initialize its `lib.IdString`. Although Version was checking for `NULL`, the lack of a string was causing problems in the output routines.
- o Fixed the FILE option which was causing the version numbers not to be displayed.

Classes/Gadget/colorwheel.gadget

- o Querying the wheel for an explicit red/green/blue value would not get the most up to date brightness value from the gradient slider causing things to get slightly out of sync.
- o Fixed bug where trying to open the class library under 1.3/2.0 would cause a system crash instead of simply failing.

Classes/Gadget/gradientslider.gadget

- o Fixed bug where trying to open the class library under 1.3/2.0 would cause a system crash instead of simply failing.

Devs/mfm.device

- o Fixed small problem of the wrong return results when opening a device with an invalid unit number.
- o Changed the format function to update the physical track and invalidate the in-memory buffer.

L/CrossDOSFileSystem

- o Booting off a CrossDOSFileSystem formatted disk does not crash an IBM machine (including CrossPC). It now presents a message "CrossDOS non-bootable disk!".
 - o Now supports formatting a compatible MS-DOS hard disk partition. It is still not bootable.
 - o Now prevents an MS-DOS hard disk partition to be formatted if not properly partitioned first.
 - o Fixed problem with method of reporting results when asked to read or seek past end or beginning of file. Now follows V37/V39 FFS method.
 - o Fixed problem with parsing a file or directory name with NULLs instead of the usual SPACES for blanks. This occurs in PC-DOS not in MS-DOS.
 - o Fixed bug that translated extended ASCII characters incorrectly when either the INTL or DANSK translation tables were selected.
 - o Added stack size checking.
 - o Fixed a bug that appeared when using the 'assign' command with this FS. A DOS deadlock could occur in certain circumstances. Added a delayed volume node addition and deletion algorithm.
 - o Fixed a small problem of freeing memory allocated by the process spawned by the Filesystem status process. This could cause memory shared to be reused when it was still needed. The fix was to add the memory entry list to the filesystem status process which is the last process to exit. This only occurs if the target device could not be opened.
 - o Fixed a problem that never showed up yet. Internally reproduced only.
- #### L/port-handler
- The main reason for this release is to fix problems that PRT: was having with Envoy printing.
- o Now has a version string.
 - o Now requires V37 ROMs.
 - o Now supports ACTION_IS_FILESYSTEM and ACTION_FINDUPDATE.
 - o Once loaded, it remains in memory permanently which fixes many problems. For example, if any of SER: PAR: or PRT: had been active, double-clicking on Format, Diskaln, or CrossDOS would cause Enforcer hits, or crashes on a machine not running Enforcer. This also fixes "copy too to PRT:" when PRT: is sending data to the envoyprint.device.
 - o When first loaded in memory, the handler patches its seglist pointer into the DOS device node of other related handlers. That is, if the handler is started as PRT:, it patches its seglist into the DOS nodes for SER: and PAR:. This means that all these devices end up sharing the same seglist for the port-handler, which can save memory and disk-loading time. It patches its seglist in any device node having "L:port-handler" as handler name.
 - o This port-handler replaces both the standard port-handler on the Workbench, and the A2232 port-handler. As such, the handler supports being activated through a mountlist entry, and will process the BAUD, CONTROL, DEVICE, UNIT, and FLAGS keyword from a mountlist.
 - o Because this port-handler replaces the A2232 one, it is possible to specify the baud rate and control flags when opening a serial unit. For example: Open("SER:9600/8N1",...);

L/FileSystem_Trans/#?

- o Modified a few values for the INTL and DANSK translation tables to default to the SPACE character for untranslatable characters instead of NULLs.
 - o Added a Mac translation table. This lets an Amiga read Mac ASCII files. This translation type can be selected from the CrossDOS commodity.
- #### Libs/asl.library
- o In screen mode requester, fixed formatting string so that horizontal scan rates with a decimal value less than .1 now come out right. That is, 29.02 was coming out as 29.2.
 - o Fixed bug where trying to open the library under 1.3 would cause a system crash instead of simply failing.
 - o Selecting Restore in the screen mode requester now correctly resets the "Overscan Type" and "AutoScroll" gadgets.
 - o Fixed bug where the file requester's drive LED would remain lit after creating a new directory when in save mode.
- #### Libs/commodities.library
- o Fixed bug present since V36. Input events of type IECLASS_NEWPOINTERPOS did not get their extended data properly copied in various places, including through AddEvents(). This resulted in the copy of these input event containing pointers possibly pointing to garbage.
 - o Fixed bug where NEWPOINTERPOS was not considered a valid class by ParseIX().
 - o Input events inserted into the input food chain as a result of the action of a translator object are now marked with the same time stamp as the input event which activated the translator object.
 - o DiaposeCkMag() was accessing a global list without proper locking. This was quite nasty, since the list is typically accessed from both the commodity input handler, and from commodity programs running asynchronously.
 - o Now nulls out the ie_NextEvent field of input events it copies. All input events generated by the library are copies of other events, and the ie_NextEvent value was left set to the original event's value. That means commodity was spawning out a lot of input events with bogus ie_NextEvent fields. This could cause crashes and confusion.
 - o InvertKeyMap() now sets the ie_SubClass and ie_TimeStamp fields to 0 in the input events it generates. These were being untouched which means they could contain garbage.
- #### Libs/iffparse.library
- o Fixed bug where trying to open the library under 1.3 would cause a system crash instead of simply failing.
- #### Libs/68040.library
- o During the setup of the MMU tables, if an expansion card was 0 bytes in size it would cause the MMU table setup to fail. 68040.library would continue to work but the magic MMU setup that would have been needed for the Zorro-III boards would not be done. This has been fixed.
- #### Monitors/#?
- o Now use a new algorithmic approach to generate the database entries. The result of this is that the amount of disk space taken up by these new monitors is 140 blocks, compared to 176 blocks they used to take.

More important is that the dimensions of Multiscan, Euro72, Super72, DblNTSC and DblPAL are now greater than they used to be. This is especially important for DblNTSC and DblPAL, which are meant to resemble NTSC and PAL as closely as possible. The Dbl.... monitors are now 720 pixels wide for MaxOScan, compared to 724 for NTSC/PAL, and the 676 they used to be!

Also, if not running with VGOOnly, the "dead" part of the display on the left hand side of the screen is now useable as VideoScan.

- o Defined ScanDoubled versions of the Multiscan, Euro72 and Super72 monitors for better coercion on AA machines.

- o All these monitors should now work under ECS, and give even greater dimensions.

- o No AA modes should be in RAM on non-AA machines.

- o Changed the names of the monitors from DoubleNTSC/PAL.monitor to DblNTSC/PAL.monitor to be consistent with the ModelID name strings.

- o All the monitors now work only with graphics.library V39.

- o With Kickstart 39.106 and VGOOnly, DblNTSC/PAL screens can only be 704 pixels wide max, whilst under the current kickstart they are 720 pixels wide. The DblNTSC/PAL monitor now checks for this.

- o Cleared up an enforcer hit with adding the A2024 after iprefs was run, and promotion was enabled.

- o Sprites were not being clipped properly in PAL A2024 modes when they crossed the panel boundaries. This was causing screen trashing when the pointer was at certain positions because the pointer would end up in the A2024's special control line.

Prefs/IControl

- o Removed "Preserve Colors" gadget. The "Avoid Flicker" gadget has been moved into the "Miscellaneous" gadget group instead of being by itself in the "Coercion" group.

- o If there is no prefs file, mode promotion is now turned on by default.

Prefs/Input

- o Now uses GTLV MakeVisible tag to ensure the currently selected keypad is visible in the listview.

Prefs/Overcan

- o On the edit screen, changed the label of the left gadget from "Use" to "OK".

- o Added code to support the new display positioning control. When you enter the overscan editing screen, if the mode supports it, there are four arrows that cause the entire visible portion of the display to move around. This is a wonderful new feature for DblNTSC, allowing sync-dependant centering of display modes. It is also possible to move the display by using the cursor keys.

- o Changed the background color on the edit screen from black to dark grey. This makes it much easier to detect when the display is being pushed too far off the right edge of the display, as the color of the display turns quite dim.

- o Made the edit screen SA Exclusive in order to avoid folks dragging it down and revealing their WB. Since the program now makes dynamic changes to haync and vaync, the WB can look bad when the edit screen is pulled down.

- o Made the edit screen SA_Interleaved.

Prefs/Palette

- o When incapable of displaying a colored color wheel, the program now puts up a gadget labelled "Color Wheel..." in place of the actual color wheel. Clicking on this gadget causes the color wheel to come scrolling up from the bottom of the screen until it fills the whole display. This color wheel screen lets the user edit the current color or reject the wheel.

There is a pair of "OK/Cancel" gadgets to accept or reject the new color selection. Clicking on either gadgets causes the color wheel to scroll off the bottom of the display. Kinda neat :-)

- o The sample section of the program now shows a sample screen title bar.

- o When opening on a custom screen, no longer has a window border and title bar.

- o When switching modes from 4 to multicolor and back, will no longer present an empty window if it is not capable of creating its new set of gadgets. The program will exit instead.

- o Selecting a color > 4 in the main palette, and switching to 4 Color Settings would not correctly redraw the gradient slider to use a color within the available selection.

- o Now uses color conversion routines from colorwheel.gadget instead of having inline duplicates.

- o Fixed incorrect flags set in some menu items so that click select of these items worked incorrectly.

- o Was letting you select between 4 and Multicolor Settings, even when running on a 4 color screen incapable of displaying the multicolor settings.

- o When running on an A2024 display, it no longer displays a color wheel, nor lets you display one.

- o Clicking in the sample section of the program causes the pen associated with the item clicked on to become selected in the pen list. That is, clicking on the sample window title bar will automatically select the "Active Window Title Bars" pen within the pen listview.

Prefs/Printer

- o Now uses GTLV MakeVisible tag to ensure the currently selected printer is visible in the listview.

Prefs/PrinterPS

- o The default margins for graphics printing are now 1 inch on the left and right, instead of 1 inch on the left and 2 on the right (this was due to a typo).

- o When in the Graphics Scaling page, selecting "Last Saved" from the menu would not cause the sample graphics to be redisplayed with the new values.

Prefs/ScreenMode

- o Fixed formatting string so that horizontal scan rates with a decimal value less than .1 now come out right. That is, 29.02 was coming out as 29.2.

System/Format

- o From within the device selection listview, it was possible to double-click on two different devices, and have Format accept the selection. A check is now done so that both clicks of a double-click occur on the same device.

- o When formatting from Workbench, Format now ignores any trailing colons in the volume name. It would get all confused if someone tried to name a disk "Hello:" instead of simply "Hello". This fixes the most common error automatically.

Tools/Calculator

- o Now gets the window title string from the current catalog instead of having it hardcoded to "Calculator".

Tools/Commodities/ClickToFront

- o Now also filters mouse clicks that come through as IECLASS POINTERPOS and IECLASS NEWPOINTERPOS, in addition to IECLASS RAMMOUSE. This fixes the bug where ClickToFront did not work with events generated by tablet drivers.

Tools/Commodities/CrossDOS

- o Fixed bug where the window would not open when there was no L:Filesystem_Frags directory, or no file within it.
- o Fixed incorrect cleanup path in case of errors while scanning directories. The bug would cause a crash the next time the window was opened.
- o Now only notifies CrossDOS handler tasks of changes in settings when new settings differ from the old ones.
- o No longer has the CrossDOS semaphore locked when it locks the DOS device list. This could potentially cause deadlocks.
- o Fixed bug where translation files were opened and read, but never closed.
- o Fixed bug that would cause a crash under V37 or harmless Enforcer hits under V39. To reproduce, start the CrossDOS commodity and copy a large file to PC0 (devs:Kickstart for example). While the copy is in progress, pick PC0 from the CrossDOS commodity and change its text filtering setting. Now click the close gadget of the commodity program. The window stays open until the large write operation completes. As soon as this happens, the window closes, and a crash occurs under 2.0.

Tools/Commodities/FKey

- o When there was a key sequence in FKey with a command such as "Insert Text" with a string associated with it, switching the command to something else, and coming back to "Insert Text" would cause the current string to be "forgotten". To get it back, you had to click in the text gadget and press RETURN. This is now fixed.
- o Selecting a command such as "Insert Text", then typing in a string as argument would cause that string to be bound to the key sequence forever. That is, even if the command was switched to something like "Cycle Windows", the string argument would be saved out to disk and reloaded in memory the next time the program ran. This was incorrect as the command didn't have anything to do with the string. String arguments are now discarded for those commands that don't support them.
- o The active key sequence is now correctly highlighted in the listview.

Tools/HDTToolBox

- o Now supports Host ID in "Partitioning" screen. Now multiple machines can share a single disk using different partitions.
- o Added CHECKBOX KIND gadget for allowing a block size other than 512 bytes in "File System Maintenance". To change file system block size in "File System Characteristics", select this gadget first.
- o Changed from BUTTON KIND gadget to CYCLE KIND gadget for "File System Type". They get labelled with their dostype as found in the file system resource list.
- o Added CHECKBOX KIND gadgets for "Fast File System", "International Mode" and "Directory Cache".

- o Added CYCLE_KIND gadget for file system block size.

- o Recognizes the "Help" key now.

HDSup/HDSup

- o Localized the name of the "Work" partition following request from Germany. Install/Install
- o No longer copies the A2232 port-handler from the install disk when an A2232 card is detected. The 39.1 port-handler handles the A2232 automatically

Appendix C: V39 ROM Changes

Following is a description of most of the important changes made to the ROM-based system software between V37 and the initial release of V39. This doesn't cover changes made to modules discussed in other talks, such as graphics and intuition.

BootMenu

- o Brand new interface featuring 4 different displays:

Main Page: Lets you select between "Boot Options...", "Diagnostic...", "Display Options...", and "Expansion Board". Clicking any one of these brings up the corresponding page. The "Boot" gadget resumes the boot operation using the options selected in the other pages, and the "Boot With No Startup-Sequence" gadget does the same, except it doesn't execute the startup-sequence.

Boot Page: Lets you control boot-related options. The listview on the left lets you pick which devices to boot from. The one on the right lets you enable/disable devices in the system. There is also a "Disable CPU Caches" gadget. It turns off the CPU caches for the current boot, which saves a lot of games that break on 68040 processors because of the big caches. Use or Cancel bring you back to the main page.

Gfx Page: Lets you pick what mode to boot in, and what chip set to emulate. Use or Cancel bring you back to the main page.

Diag Page: Lists all the boards in the system and shows their state (Working or Defective). This page is automatically brought up during system bootup if a board is defective.

- o Hitting any key toggles the display between NTSC and PAL. A message on the main page indicates this fact.

cia.resource

- o Changed the priority of the interrupt servers to +120 such that they don't miss interrupts.

con-handler

- o Fixed a low-memory trashing problem where CON: would signal a NULL task.
- o It no longer will use proportional fonts (or rather fail at trying to use them) when opened on a public screen.
- o It is no longer possible to size the window such that the entire interior goes away.

- o Fixes the title bars (the parsing routine wasn't skipping over delimiters). This also fixes the "funny" results you would get with malformed CON: lines (like con:0a/0/640/200/title having a title of "200").

- o Fixes negative sizes in the window spec. Height < 0 gives you max displayable height.

- o Always opens window big enough for one line of text.

console.device

- o CMD_CLEAR fixed, was broken in V37.
- o Uses screen dimensions (rounded off to nearest byte width) to calculate maximum character map width instead of using bm_BytesPerRow. Likewise uses screen height instead of bm_Rows.

- o First pass at the scrolling optimizations. Recalcs scroll mask at reset time (set to defaults), at full clear screen time (FF, or HOME/CLS), and whenever you set new pen/cell/background colors via the standard SGR sequences.

- o Rework how conceal mode works - no longer sets rp_Mask to 0 which also disabled ALL output, including scrolling, clearing, etc... not good for character mapped consoles in particular since it causes the display & map to lose sync.

- o Scroll DOWN no longer tries to fill in vacated portions of the window with text in the off-screen buffer (if any). This was useless as a scrollbar feature, could crash if you scrolled down more than one window's worth of text (ouch), didn't perform a window refresh (thereby leaving hidden characters in the visible map until you resized, or revealed), and was inconsistent with SMART REFRESH consoles - the application using scroll down would reasonably expect that the vacated portion is entirely empty.

- o First pass at breaking up large CMD_WRITEs. Now unlocks layers approx every 256 characters (this is simple, low-overhead code which does not try to be exact). Recalcs everything when layers are relocked. Helps quite a bit. No more locks for the entire file when doing COPY foo *. Makes console much more friendly; you can now resize windows (only minor delay) during long writes, click in other windows, etc.

Note this simple code won't work for a long stream of text with no control characters; this however is extremely rare. Even text files have LF's, however in any case the problem is no worse than it was before.

- o The above feature makes it possible to cheaply break a CMD_WRITE, hence making it possible to easily fix the DisplayBeep() deadlock bug. DisplayBeep() is now postponed a few CPU instructions until after layers are unlocked.

- o First pass at modifying mouse tracking code (drag selection) to use input events instead of PeekQualifier().

This code change was added so tablet drivers, and commodities can be used in character mapped consoles with selection capability.

PeekQualifier() returns only what input device thinks the qualifiers are; not what's seen by applications which watch/use the input stream.

- o Now supports a new private sequence ("ESC| s") which sets the current SGR settings as your defaults. This affects ESC[0m (reset all SGR settings), ESC[39m (reset default primary pen), and ESC[49m (reset default secondary pen (cell color)).

Text style info, and reverse mode (on/off) are also saved, and hence restored when ESC[0m is sent.

This is intended as a user sequence for use in your shell-startup; it allows you to use other colors/settings, and not have these constantly reset by programs like MORE, LS, etc. I'm recommending it not be used by applications; only users for their shells. An application which can deal with this problem of SGR settings should continue to do whatever it is doing now. ESCC (reset console) does however reset the default SGR settings to their true defaults.

- o Downcoded pack.c. Is many times faster (if the maps are not disorganized; the maps become disorganized as text is scrolled off screen, so in these cases an initial resize can still take a moment - didn't want to touch that code though). For an organized map, resizing even very large windows (e.g., Monterm size) with 8x8 or smaller fonts (so we have a really large map) is virtually instantaneous on a 3000, and nearly so even on 68000 machines. It still takes time to redraw

the text (limited primarily by the Text() function), but the time needed to pack, and unpack the map is a fraction of what it was.

- o Borderfill code added so ESC|>8m fills to borders if an explicit line length and/or page length have not been set. No change to cu_xrExtant or cu_yrExtant in public portion of console unit structure. Border refers to the area outside of the normal console rendering area up to the window right/bottom borders. The size of this area is 0-N pixels where N is a maximum of the font width-1 or height-1.
 - o OpenDevice() now fails if trying to open a character mapped console, but memory can not be allocated for the map. In V37, OpenDevice() returned success for this case which left you with a half functioning console window - clearly confusing for the user, and virtually worthless because of the lack of refresh info needed to fix up damage.
 - OpenDevice() still works the same if you have a SIMPLE REFRESH window, but did not ask for a character map (uncommon, but doable).
 - o Also fixed a bug which you may never see now that the above code was added: clearing a simple refresh window which lacked a character map cleared a garbage rectangle; layers prevents this from being a crash, and code elsewhere inhibited negative rectangles. The bug exists in 1.3 also, and was partially fixed for 2.0; the bug use to be apparent in SUPER_BITMAP windows, and because the case of SIMPLE REFRESH without a map is rare for console.device, the bug has probably never been noticed (found during memorization testing!).
 - o Removed code which checks to see if the application had drawn over the cursor in a console.device opened by the application. The kludge did a ReadPixel() of every pixel where the cursor was drawn, and if any bits did not match the expected color (also modified by pattern), cursor drawing was turned off for that console for as long as that console window was opened. Applications (few) which draw over the console cursor, but do not explicitly turn it off will now have a patterned rectfill the size of the cursor (generally 8x8) in the upper left hand corner of the window if the window is deactivated. This is a minor visual problem, though not one which should cause anyone to crash, or not run. The problem will also never be noticed if the console window is not deactivated.
- dos.library
- o Fixes a bug in Open() where if the path was more than 127 characters long, a random byte of memory would be trashed.
 - o Has support for fib_OwnerXXX for the networking people. ExAll() supports ED_OWNER.
 - o Added ExAllEnd().
 - o Added SetOwner().
 - o Fixed overrun error in FGets() (if no newline or EOF, it reads one byte too many into your buffer - workaround for V36/37 - allocate buffer 1 byte larger than passed in).
 - o HUNK_RELOC32SHORT now works at the right value (1020). Also added a 32-bit PC-relative reference mode, mainly for >= '020-only executables.
 - o Added GVF SAVE VAR. For SaveVar(), it will now do the same actions for ENVARC\$whatever, as well as ENV\$.
 - o Fixed character classes in ParsePatternNoCase(). The classes weren't being promoted to upper case (i.e. [a-z] should have become [A-Z]). Note that only ParsePatternNoCase() was affected by this bug, not MatchFirst()/MatchNext()

- o Fixed FreeDosObject(xxx,DOS_CLI). It wasn't freeing the strings associated with the CLI.
 - o FindCliProc(0) now returns NULL, even though it's an invalid CLI number.
 - o For people passing in an RDArgs structure but no RDA_Buffer to ReadArgs(), it now properly clears out RDA_Buffer on FreeArgs(), so you can reuse it safely (without having to clear it out yourself).
 - o StrToLong() was returning the number of white-space characters if no digits were found. It now properly returns -1.
 - o Fixed GVF_DONT_NULL_TERM for global variables.
 - o The initial console window on bootup now opens the size of the Workbench DCLIP.
 - o Fixes the CliInitNewCli() open of S:Shell-Startup when no FROM file is specified.
 - o Fixed "Copy CONSOLE: foo" by making GetDeviceProc() know about CONSOLE:.
 - o Localized "Software Failure".
- filesystem
- o Added support for DOS\4 and DOS\5 file systems which offer directory caching. DOS\4 and DOS\5 are orders of magnitude faster at directories than other file systems, since they keep a cached copy of all the ExNext()/ExAll() data appended to the directory. This does require a few extra block accesses on create and delete, and also after modifying the file (in Close()). Create speed dropped about 30%. However, directory speed is 7-20 times faster.
- For floppies, this usually means that dir or list take about 3/4-1 second to start, and then you get most or all of the directory instantly, or within 1/2 second or so (it may take 1 or 2 seconds for really big directories).
- o Fixed a bug with delete for non-DOS\5 partitions.
 - o Fixed a random memory trash in the filesystem in a race condition. When two renames hit just the right timing, one has to wait on the other, and the wait routine used the wrong (garbage) register to get the head of the list. This trashed 1 longword of semi-random memory, and then hung the rename forever.
- gadtools.library
- o LayoutMenus() and LayoutMenuItems() recognize some new tags to support NewLook menus:
 - GTW NewLookMenus (BOOL): requests NewLook menu treatment.
 - GTW Checkmark (struct Image *): checkmark you'll use in menus
 - GTW AmigaKey (struct Image *): Amiga-key image you'll use in menus
- Basically, if you open your window with WA_NewLookMenus, also lay out your menus with GTW NewLookMenus. If the menu-item font will be the screen's font, that's all you need to do. If the menu-item font is something else, you must create a checkmark and an Amiga-key image, and pass each one to both Intuition (WA_Checkmark and WA_AmigaKey) and to GadTools (GTW_Checkmark and GTW_AmigaKey).
- GTW FrontPen is now recognized. If GTW NewLookMenus is specified, this attribute defaults to the screen's BACKDROP_PEN, else it defaults to "do nothing", which allows the GTW_FrontPen tag that may have been passed to CreateMenus() to still hold.
- o STRING_KIND, INTEGER_KIND, and BUTTON_KIND gadgets now support the GA_Immediate tag.

- o You can now put an arbitrary command string in the right-hand side of a menu, where the Amiga-key equivalent goes. To do this, point the `NewMenu nm_CmdKey` field at the string (eg. "Shift-Alt-F1), and set the new `NM_COMMANDSTRING` flag in `nm_Flags`.
- o If a window has multiple checkboxes or radio buttons, a separate image is no longer allocated for each one.
- o The bevel box of the slider and listview now refresh with the gadget, instead of with `GT_RefreshWindow()`.
- o Scrollers with `GA_RelVerify` set weren't sending `IDCMP_GADGETUP` messages when the arrow buttons were released.
- o `GadTools` now uses `SetABPenDrMd()` when advantageous.
- o New `GTMX_Scaled` and `GTCB_Scaled` tags instruct `GadTools` to scale the `mx` button and checkmark respectively to the dimensions supplied in the `NewGadget ng_Width` and `ng_Height` fields. Under V37, or in the absence of these tags, the dimensions are fixed. Added `$defines` for those dimensions.
- o `GadTools` now has a `GT_GetGadgetAttrA()` function (and a `GT_GetGadgetAttr()` `varargs` version). This function retrieves attributes of the specified gadget, according to the attributes chosen in the tag list. For each entry in the tag list, `tl_Tag` identifies the attribute, and `tl_Data` is a pointer to the long variable where you wish the result to be stored.
- o Checkboxes now return their "selected" state in the `IntuiMessage->Code` field.
- o Many new tags for `CreateGadgetA()` were added:
 - `GTTX_FrontPen` and `GTTX_BackPen` to let the color of `TEXT_KIND` gadgets be controlled.
 - `GTNM_FrontPen` and `GTNM_BackPen` to let the color of `NUMBER_KIND` gadgets be controlled.
 - `GTNM_Format` to specify the formatting string to use with `NUMBER_KIND` gadgets. This is so a localized number format using "kld" instead of "dld" can be used.
 - `GTNM_MaxFormatLen` to specify the maximum length of the string that can be generated by `GTNM_Format`.
 - `GTTX_Justification` and `GTNM_Justification` to allow for right and center justification on `TEXT_KIND` and `NUMBER_KIND` gadgets.
 - `GTSL_MaxPixelLen` lets you specify the maximum pixel length the level display of a `SLIDER_KIND` gadget will occupy. This allows proportional fonts to be used with sliders.
 - `GTSL_Justification` specifies how the level display of a `SLIDER_KIND` gadget is to be justified within the width allocated by `GTSL_MaxPixelLen`.
 - `GTIV_MakeVisible` for listviews. You pass it an item number and it makes sure it is visible within the listview display.
- o Many new tags for `GT_SetGadgetAttrA()`:
 - `GTTX_FrontPen`, `GTTX_BackPen`, `GTNM_FrontPen`, `GTNM_BackPen`, `GTNM_Format`, `GTTX_Justification`, `GTNM_Justification`, `GTSL_Justification`, and `GTIV_MakeVisible` all have the same purpose as described for `CreateGadgetA()` above.
 - `MX_KIND` gadgets now support `GA_Disabled`.
 - `GTSL_DispatchFunc` and `GTSL_LevelFormat` are now changeable via `GT_SetGadgetAttrA()` instead of being create-time only attributes.

- o Added the `GTBB_FrameType` tag which gives access to the new frame types available in Intuition. You pass the tag to `DrawBevelBox()` and can specify `BBFT_BUTTON`, `BBFT_RIDGE` or `BBFT_ICONDROPOBOX`.
- o `GT_SetGadgetAttrA()` can now safely be called when the gadget being affected is not attached to a window, by passing a `NULL` window parameter.
- o Specifying `GTTX_CopyText` and not `GTTX_Text` now works for `TEXT_KIND` gadgets.
- o `TEXT_KIND` or `NUMBER_KIND` gadgets that have the `GTTX_BackPen` or `GTNM_BackPen` tags specified look visually cleaner when changing the gadget text using `GTTX_Text` or `GTNM_Number` than those without. Listviews take advantage of this when applicable.
- o The value of `GTSL_Level` is now bounds checked at `CreateGadget()` time in addition of at `GT_SetGadgetAttrA()` time.
- o Fixed bug where doing `GT_SetGadgetAttrA()` on a `MX_KIND` gadget and not passing the `GTMX_Active` tag would reset the active selection to 0 instead of leaving it alone.
- o The `NewGadget.ng_TextAttr` field can now be `NULL` whenever a gadget is created. In such a case, the screen's `TextAttr` is used (screen's `TextAttr` is determined from the `VisualInfo` in the `NewGadget` structure).
- o The level display of `SLIDER_KIND` gadgets is now rendered with background set to `BACKGROUNDPEN` instead of 0, which is more "correct".
- o Now uses `SetWriteMask()` instead of `SetWrMask()`.
- o Added support for gadget help in all gadget types.
- o `GadTools` now handles the new `ExtIntuiMessage` generated by Intuition.
- o Many enhancements to `PALETTE_KIND` gadgets:
 - Palette gadgets no longer display a box filled with the selected color. The selected color is instead denoted by a box drawn around the color square in the main palette area.
 - Palette gadgets now allow strumming, and right mouse button cancellation.
 - `GTGA_ColorTable` is a new tag to support sparse color tables in `gadtools`. This tag can be passed at `create/set/get` time.
 - `GTGA_NumColors` is a new tag to specify the total number of colors to display. This allows amounts of colors that are not powers of 2. This tag is also good at `create/get` time.
 - `GTGA_ColorOffset` is now supported at `get/set` time.
- o Renders itself much faster, this makes a big difference on 256 color screens.
 - Now does quite smart layout of the color squares. An attempt is made to keep them as square as possible, based on the aspect ratio information obtained from the `gta` database. As many colors as possible are put on the screen, until things get too small in which case the upper colors are thrown away.
- o `MX_KIND` gadgets now support `NgGadgetText` and will display the label in relation to the group of `mx` gadgets.
- o Added `GTMX_TitlePlace` tag. This determines where the title of a `MX_KIND` gadget is displayed. If this tag is not provided, the title is not displayed. This is required for compatibility.
- o Fixed size calculation errors in listview present since V37. This may cause certain listviews to change in size from their V37 size.

- o Revamp of listviews:

- Listview lines can no longer end up complemented in certain tricky situations involving detaching lists.
- Listviews correctly track the selected line when you click in them.
- Listviews were clipping the text four pixels early on the right.
- Added GTLV Callback. This tag allows a callback hook to be provided to gadtools for listview handling. Currently, the hook will only be called to render an individual item. This adds the very useful ability to define a callback hook which is used to scroll complex items such as graphics, etc.
- Listviews now allow strumming. That is, holding down the left mouse button and moving the mouse up or down causes the active selection to track the mouse. Moving the mouse off the top or bottom of the listview causes the list to scroll.
- Listviews that used to have a display or string gadget underneath them now have a highlight bar to indicate the selected item. This is in anticipation of listview multi-selection. If the listview had a display gadget, it no longer does as the highlight bar is used. If a listview had a string gadget, it retains it.
- Listviews highlighting is done using the pen-spec method instead of the 1.3 complementing method.
- Listviews are much faster at rendering and scrolling, which makes a noticeable difference in 8 bit planes
- Added GTLV MaxPen tag to specify the maximum pen number used by a custom rendering callback hook. This enables more optimal scrolling and screen updates.
- (GTLV_Selected, -0) is now supported at both create and set times.
- (GTLV_Labels, -0) is now supported at create time.
- Listviews now support GA Disabled. This causes the list area to be ghosted, but the scroller and arrows remain unghosted.
- When cloning a rastport for internal use, no longer copies the TmpRas field of the original rastport, which should eliminate some potential bugs.
- Changed the definition of TEXTIDCMP and NUMBERIDCMP in gadtools.h to be (0) instead of (NULL), to keep the Manx compiler happy.
- It is now safe to call GT_GetGadgetAttrs() and GT_SetGadgetAttrs() with NULL gadget pointers.
- When GT_SetGadgetAttrs() is called on an active STRING_KIND or NUMBER_KIND gadget, the gadget is automatically reactivated after its string is changed. Although this reactivation flickers, the functionality is quite useful.
- Now copies a complete TTextAttr structure when needed to fix potential problems with WeightMatch(). This is only done to create underlines under gadget labels.
- Fixed example in CreateGadget() autodoc. Only had a single argument in the call to GT_RefreshWindow()
- BOOPSI images are now allowed in gadtools menus.
- Added the GTTX_Clippped tag for TEXT_KIND gadgets.

- ramdrive.device

- o Uses AllocMem(XXXX, MEMF_REVERSE|MEMF_KICK|MEMF_NO_EXPUNGE) instead of private AllocHigh() code.
- o Uses CopyMem() instead of an unrolled loop. CopyMem()'s MOVEM's are faster than the unrolled loop used previously.
- o Protects KickTag/KickMem list with FORBID/PERMIT inside of KillRad() vector used by REMAP. Fixes possible crash if some other task is fiddling with these lists at the same time.
- ram-handler
- o Fixes the long standing bug where if the file you were examining with Extent() is deleted, RAM: goes off into never-never land (and your system follows). If the file is deleted, it will restart at the beginning of the directory.
- o Disabled softlinks in RAM: to save ROM space.
- shell
- o NewShell/NewCLI now open full with (like Shell from WB).
- o NewShell/NewCLI now handle FROM fields up to 255 long (up from 127) and errors out if FROM or WINDOW are too long.
- o Resident module handling now properly Forbid()s around seg_UC+/---.
- o Prompt now handles \$*.
- o Removed two harmless enforcer hits at boottime.
- o If a "command 'command...'..." fails, it no longer inserts the error message and continues (FailAt is used to determine failure).
- o Added evil kludge to solve the problem of 1.3 SetClock crashing on 68040s.
- o Fixes the write to rom on <> redirection.
- o NewShell/NewCLI no longer print error messages if no S:Shell-Startup is present.
- timer.device
- o Timer keys off new GfxBase flag for determining EClock frequency since PAL/NTSC is now software selectable.
- trackdisk.device
- o Post-write delay has been moved from 2ms (spec is 1.2ms) to 3ms, and side settle delay from 1ms (spec is 0.1ms) to 1.5ms. This should fix most A1010's out there. In addition, both of those values have been made part of the public unit structure like settle delay and step delay, so people with REALLY bad A1010's can back it off as far as they need to (for setpatch can).
- o Fixes a nasty oversight in the HD floppy handling. After switching from a HD floppy to LD floppy and back to HD floppy, you could never safely write to an HD floppy unless you formatted an HD floppy first.
- What happened was that the extra alop area at the front of the write wasn't getting set to \$aaaaaa, it was being left with garbage from the last LD read (since LD uses less alop, it's start-read spot is earlier). Format re-init's the entire buffer, as does the first switch to an HD floppy (only the first, since it switches to a larger buffer then).
- o Fixed a bug where if a track was totally unreadable it returned the number of retries as the IO ERROR instead of TDErrorNoSechdr (this was causing the "Error 11" stuff when you popped a disk while copying from it).

utility.library

- o Downcoded all tag calls from C to assembly which yields substantially faster performance.
- o Removed 68020-specific versions of the date conversion routines.
- o Fixed bug in MapTags() where the "includeMiss" parameter didn't work.
- o Cleaned up and expanded autodocs.
- o Cleaned up public include files.
- o Added comments in the autodoc entries for the 4 32-bit math routines, to the effect that they preserve address registers, and that A6 does NOT have to be loaded in order to call the routines. This is an exception to the standard rule, but can avoid register shuffling which is important in low-level math routines.
- o Made the math routines several cycles faster on 68000 machines.
- o Added SMult64() and UMult64() which do 32x32-64 bit integer math.
- o Added ApplyTagChanges().
- o Added two new library calls that are mainly here to help intuition get smaller. These are PackStructureTags() and UnpackStructureTags().
- o Added the NameSpace code.
- o Added GetUniqueID().

Appendix D: Release 3.01 ROM Changes

Following is a description of most of the important changes made to the ROM-based system software between Release 3 and Release 3.01. This doesn't cover changes made to modules not discussed in this talk. It also doesn't cover any changes after December 1992. Since 3.01 is still under development as of this writing, more changes are likely to be made to the software prior to release.

BootMenu

- o Fixed bug where the chip type mutual exclude gadget was being displayed even on pre-ECS machines.

exec.library

- o Added the full support for the Zorro-III quick interrupts. The new IVO (in an old slot) ObtainQuickVector() is used to allocate the vector. There is no deallocation since this is basically a configuration issue and not a dynamic thing.
- o On machines with PCMCIA cards, EXEC now makes sure the interface is turned on at boot time and then will turn it off before configuration. This should let a full 8-meg of RAM be added in the Zorro-II space. This change requires an update to the credit card resource/device such that it will correctly turn on the interface if needed.
- o The Quick Interrupt vectors that have not yet been added used to be -1. Now they point at an Alert that is the new Unexpected Quick Interrupt.

gadtools.library

- o Fixed bugs in clipping code in TEXT KIND and NUMBER KIND gadgets. The clipping didn't work correctly on right and center justified text, and was under-evaluating the number of pixels available for the text in a gadget that didn't have borders.
- o Fixed bug in the calculation of the default value for the GTSL MaxPixelLen tag. This caused odd clipping of the number display for sliders whenever the title of the gadget wasn't on the same side of the slider as the display of its current value.

- o Fixed GTJ CENTER option for the various CTXX justification tags. The way centering was done could cause certain characters to get lost.

carddisk.device

- o Now flushes cache during data writes in anticipation of 040 copyback cache on Al200 (no hardware support for PCMCIA memory space data cache control provided, so the data cache is still potentially a problem when programming flash rom; means turning off the data cache globally for 030/040 Al200's to support 6-10us write/verify timing).

card.resource

- o Now leaves PCMCIA slot disabled if any RAM is configured at \$600000; this allows use of >4Megs of 24bit RAM on the Al200 at the expense of being unable to use the PCMCIA slot.
- o Partial work around for a hardware bug in our PCMCIA implementation which presents 2Meg+ addresses everytime we access ATTRIBUTE memory. This causes a problem when a >2Meg card which ignores REG is used (and a potential problem with any card which tries to decode the entire address when REG is set). The former problem is kludged around by trying to sniff out mirroring of 4 bytes at \$A00000 and \$800000 but not mirrored at \$600000.

- Tested with:

- Fujitsu 512K SRAM (ignores REG)
- Fujitsu 128K SRAM (decodes entire address)
- Panasonic 512K SRAM (returns \$FF for attribute memory)
- HP 128K SRAM (has 16 bytes of attribute memory)
- NewMedia 2M PSRAM (ignores REG)
- NewMedia 4M PSRAM (ignores REG - this is the card which demonstrates the problem)

- o Considerably faster memory sizing for SRAM/DRAM cards (does test of every 256 words/long-words) - tested with:
 - Fujitsu 512K SRAM (ignores REG)
 - Fujitsu 128K SRAM (decodes entire address)
 - Panasonic 512K SRAM (returns \$FF for attribute memory)
 - HP 128K SRAM (has 16 bytes of attribute memory)
 - NewMedia 2M PSRAM (ignores REG)
 - NewMedia 4M PSRAM (ignores REG - this is the card which demonstrates the problem)
- o CardMemoryMap structure extended for V39 card.resource. Now has COMMON/ATTR/IO Memory Zone size for lookup via structure. Will be used to provide splitting of memory zones in the future if needed. No change for existing software.
- o BVD1/SC, BVD2/DA, and BSV/IRQ status change interrupts can now be individually enabled/disabled. WR (Write-Protect) status change interrupts are always enabled (rare), and there is no change in the defaults. This is intended for future use if needed (e.g., Flash-ROM which expects software to poll SC during programming; better performance can be obtained if interrupts are not generated). If needed on the A600, this can be implemented as documented work around, or Sefunction() of CardMicControl(). No expected change for existing software; defaults are the same as they use to be in V37 card.resource. Spurious interrupts (change true, but interrupt disabled) are cleared by the resource software, and hidden from the status change callback hook.
- o Secondary callback option for status change interrupts; allows high-performance hardware to be serviced via interrupts only (instead of signalling a task).
- o Flush Cache when ReleaseCard() is called. A flush before full release ensures that no more writes will occur once the caller returns from ReleaseCard(). This is to support the 040 copyback cache when/if an 040 becomes available for the Al200. Would prefer control over the data cache for PCMCIA space independent of the first 4MEG of 24bit Fast RAM, but we don't have this feature. Lack of Data Cache control for PCMCIA space is still potentially problematic for use of FlashROM programming which requires disabling the DATA cache for 030/040 equipped Al200's so that fast (6-10us) write/read operations can be performed during programming. Disabling the DATA cache during FlashROM writes means disabling globally.
 - dos.library
 - o Fixed bug in RemAssignList(): it wouldn't remove the first lock in the assign.
 - o AttemptLockDosList() was returning NULL or 1 for failure instead of NULL.
 - o Made RunCommand() free any memory added to the tc_MemEntryList by the command being run. tc_MemEntry is now saved and emptied before calling the command, and restored after any added memory is freed.
 - o Fixed ExAll() emulation to not lose 1 file each time the list is broken up into multiple ExAll() calls.
 - o Removed broken attempted fix for rda_Buffer. Autodocs now reflect that you must restore rda_Buffer before each call to ReadArgs() if you pass in an RDAArgs structure. Now always clears rda_Buffer in FreeArgs().
 - o SetVBuf() enabled.

- o Changed some prototypes to avoid c++ reserved word "template". Changed VPrintf()/VPrintf() prototypes to VOID * from LONG * to reduce useless compiler warnings/casts.
 - o GetDeviceProc() now returns errors better (especially ERROR_NO_MORE_ENTRIES). It used to lose error codes by calling Unlock().
 - o SetVBuf() now updates the filehandle so it won't overwrite the buffer with a smaller one if SetVBuf() is called before doing buffered IO. Also it doesn't allocate anything if the new size is the same as the old.
 - o SetVar() now creates subdirectories as needed (including multiple ones) if they do not exist already (in ENV; and in ENVARC: if GVF_SAVE_VAR is set). Also, it now preserves any IoErr() and won't try to save to ENVARC if there is an error saving to ENV;.
 - o Modified to fix an edge condition which existed when making the mod to SetVBuf().
- expansion.library
- o New Al200-specific build that can detect CPU Slot RAM (\$08000000) if you have a 32-bit addressing CPU installed. The CPU slot area is 128Meg in size (just like the A3000) but has the addition of a wrap check at each lMeg of space in the CPU address space to make low-cost RAM expansion possible without jumpers. (It is now possible to get 128Meg SIMMs so a single SIMM on a CPU card could make a 128Meg of FAST RAM system)
- The reason that this has to be Al200 specific (at least for now) is that the behavior of the existing A500/A2000 CPU cards with respect to 32-bit addresses is very undefined. They act very strangely and differently making it very difficult to safely figure out if these cards are operating correctly or not.
- filesystem
- o Fixed deletion of the destination of a hardlink - this was badly broken in all versions of the FS. DCFS just made it easier to hit. This was causing spurious "Checksum Error on Block 0" errors (and potentially others), especially when UUCP was using a DCFS partition.
 - o Fixed a return code which would make softlinks not work if a softlink to a directory is in the middle of a path.
 - o Fixed the buffer overrun on ExAll() with ED_COMMENT if the first character was >\$80 (and lost the first character of comments).
 - o Fixes updating the date of a directory that changes in the parent of that directory's dircache.
 - o There were old offsetting bugs in the ExAll() filename/comment copying code. When I fixed the code not to copy too many bytes, the clear was being done to the wrong byte.
- filesysres.resource
- o Now matches the FS version change.
- workbench.library
- o Adjusted the sizes of the OK/CANCEL and SAVE/CANCEL gadgets in the Workbench requesters to match the rest of the system.
 - o Fixed a long standing bug that was just found: The system would crash (sometimes) or cause Enforcer hits if files were deleted within a drawer that was also selected for deletion. This one has a fundamental flaw in Workbench which had to be patched with some rather tricky organization of tests...



IFF

by Carolyn Scheppner

ILBM Files: V39 and AA Support Issues

For compatibility with and support of enhanced Amiga graphics capabilities, both the short-term and long-term possibilities, you must modify your software to remove any built-in limitations which would prevent you from growing *with* the Amiga.

Get *Rid* of Hardcoded Limits, Write Software that Adapts

Many of the Amiga graphics software packages currently on the market are hardcoded like the old "DF0: DH0:" file requesters.

Such hardcoded graphics application software limits include:

- ☐ offering a fixed set of display modes or sizes
- ☐ offering a fixed range of depths or sizes for certain display modes
- ☐ loading or handling a maximum of 32 colors
- ☐ dealing with color guns as 4-bit values

The first thing you need to think about when upgrading your application for V39, is *not* to upgrade it for V39. You must upgrade your software so that it can adapt to arbitrary display modes, depths, and sizes.

If you offer different display modes, do not arbitrarily restrict the modes that you offer. If 8-bitplane hires, or hires HAM, are supported by a new chip set, and your software restricts a user to 5-bitplane hires and lores HAM, then your software will be obsolete.

Rewrite your software to use features such as the 2.1 display mode requester. Make sure that your software can adapt to larger palettes. Handle R G and B values internally as at least 8-bits, not 4.

Proper IFF ILBM Support

When loading, saving, and processing ILBM files, care must be taken to properly support the

current and future graphics capabilities of the Amiga. Hardcoded limits and assumptions often exist in ILBM handling code. The NewIFF39 code modules attempt to properly implement all of the following concepts in a backward (and hopefully forward) compatible manner.

Proper ILBM.CAMG Chunk

Saving

If running under V36 or higher, save a 32-bit mode id in the CAMG ULONG. Some of these modes have all zeros in the upper word and are classic Amiga viewmodes. Others are more complex but generally provide a good bit pattern in the low word to allow reasonable results if displayed with an old viewer. You may wish to let your user decide if a different mode id should be saved. For example, a user may prefer to work in DBLNTSC but need to save his images as plain Hires LACE for genlocking. See V39 graphics/modeid.h for current modes. Use the 2.1 ASL screen mode requester or the display database if you wish to offer only all modes available on the user's system. Use ULONG modeid = GetVPMODEID(viewport) if you are saving the mode id of a display.

Loading

Support reading and using full 32-bit mode ids from CAMG, with some screening for bad IDs, and fallback code if ModeNotAvailable(). Screening is required because there are some CAMGs out there with garbage in the upper word. See sample "getcamg" code at end. Under V39 or higher, the new BestModeIDA() graphics function can be used to query for a suitable and available replacement mode when an ILBM CAMG mode is not available. Stop looking at the bits of graphics mode ids. See the NewIFF39 code modules for example usage of BestModeIDA().

Proper ILBM.BMHD X and Y aspect

Saving

Use the display database in a simple manner to get the correct x and y aspect values for the ILBM.BMHD. See the "getaspect" code below. This code gets the correct aspect ratio for any viewport modeid from the display database. If running under < V36, it falls back to updated 2.0-compatible aspect values for old modes.

Loading

Perhaps you can start to expect reasonable information in the ILBM.BMHD x and y aspect fields.

Proper 8-bit-per-gun ILBM.CMAP

Problem: the CMAPs of many ILBMs contain only 4-bits-per-gun of R, G, and B, each left justified in a CMAP byte - for example, white saved as \$F0 F0 F0 rather than \$FF FF FF. This made no difference when Amigas could only display 4-bits-per-gun of color (i.e., 4096 different colors) since only the upper 4 bits of each CMAP byte were used when setting colors, and for example, RGB of \$F F F, whether stored as \$F0 F0 F0 or \$FF FF FF, was 4-bit white. But AA Amigas can display 8-bits-per-gun of color (16,000,000 different colors), and \$F0 F0 F0 is not quite white - \$FF FF FF is. To properly display 8-bits of color from an ILBM whose CMAP contains only 4-bits of color information per gun, you must know to scale the 4 bits to 8 bits. But in most ILBMs there is no flag to tell you whether the CMAP contains 4 left-justified or 8 significant bits per gun.

Here are tips on saving and loading 8-bit-per-gun colors, and some strategies and a new flag bit for recognizing and dealing with both 4-bit and 8-bit CMAP colors.

Saving

Either always save CMAPs with 8 significant bits-per-gun, or save 8-bits-per-gun by default and offer a 4-bit palette save option (for compatibility with some products which may contain containing broken handcrafted CMAP color handling code). When saving from a 4-bit-per-gun source, do *not* left justify each 4-bit gun in the CMAP R, G, and B bytes, but rather SCALE each 4-bit value to 8 bits by duplicating the 4-bit value in the upper and lower nibble of its R, G, or B CMAP byte (e.g., save white as \$FF FF FF, etc.). When saving under V39 or higher, if you must extract the colors from a display, use the new 32-bit graphics color getting function GetRGB32(). Do not read a ColorMap's ColorTable directly. Save the 8 most significant bits of each gun in the CMAP.

Important: A new advisory BMHD flag, BMHDF_CMAPOK, has been defined to indicate that an ILBM contains an 8 bit significant CMAP, not an old 4-bit left-justified CMAP. The advisory flag is the high bit (1 << 7) of the BMHD.Flags byte (formerly named BMHD.Pad1 or BMHD.Reserved1). Whenever you save an 8-bit-significant CMAP (either 4 bits scaled to 8 bits or true 8 bits), we suggest that you set this flag bit in your BMHD.Flags to advise aware loaders that your CMAP values are definitely not 4-bit shifted values and do not need scaling to 8 bits.


```
#define BMHDB_CMAPOK      7
#define BMHDF_CMAPOK      (1 << BMHDB_CMAPOK)
```

Loading

Detect and use all 8 bits of CMAP color, if provided, when running under V39 or higher machine. If you see the new BMHDF_CMAPOK flag set in BMHD.Flags (described above), then you can be sure that the CMAP already contains 8 significant bits per gun of color. If you are running under V39 or higher, scale each 8-bit gun value to 32-bits (by duplicating it in all 4 bytes of a ULONG), and load the colors using one of the V39 32-bit color loading functions (LoadRGB32(), SetRGB32(), SetRGB32CM()).

If the BMHDF_CMAPOK bit is not set in the BMHD, then to display the CMAP colors correctly on an AA system, you need to determine if the stored CMAP color gun values are shifted 4-bit or full 8-bit values. If you do not examine the CMAP and instead just assume that it has 8 significant bits, you will display the colors of many ILBMs incorrectly dim, while older LoadRGB4() loaders will actually display the correct colors (because the V39 4-bit color loading functions will scale the passed 4-bit values properly to 32-bits).

You can try to determine if the CMAP contains all 4-bit left-justified RGB values by examining the low nibble of every CMAP entry you intend to use (do not examine additional registers or garbage which may be stored in the CMAP). If every examined low nibble is 0, then you would probably be correct to assume that the CMAP contains 4-bit left-justified values. In this case, you can correctly scale these values to 32-bits by first scaling to 8 bits (i.e., duplicate the upper nibble of each gun into its low nibble), then scaling to 32-bits (as described above). If any of the examined CMAP nibbles is non-zero, then the 8-bit CMAP values are directly scaled to 32-bits. Then load the colors using one of the V39 32-bit color loading functions.

When loading on a pre-V39 machine, just use the upper (most significant) nibble of each 8-bit CMAP gun value when calling the old 4-bit-per-gun pre-V39 graphics functions (LoadRGB4, SetRGB4, SetRGB4CM). Be very careful to AND out the low nibble of each gun before shifting and OR'ing R, G, and B guns to create an xRGB word for pre-V39 functions.

Stop Limiting Color Register Load Counts to 32

Older IFF code, and even the early V37 NewIFF code would read any number of colors from an ILBM CMAP, but would only set a maximum 32 colors in the display. Instead, the maximum number of colors set in the display should be limited by the display Viewport's ColorMap->Count rather than a hardcoded limit. The current V37 and V39 NewIFF have no fixed limit on number of color registers.

Stop Limiting Depth to 5/6

Older IFF code had fixed limits for the maximum allowable depth for displays and ILBMs. Remove your limits. Display as much as the system can handle. Don't reject depths and depth/mode combinations arbitrarily. Also, you may want to stop assuming that a 6-plane ILBM with no CAMG is HAM or HALFBRITE (although that might still be a good assumption since only a pretty lame program would write a HAM or HALFBRITE ILBM with no CAMG chunk).

Watch out for BitMap->BytesPerRow

We have seen a number of products which have problems with BitMap->BytesPerRow rounding changes under V39 with AA.

BitMap->BytesPerRow is a modulo - it is the number that must be added to the address of a bitmap byte to get to the same byte one scan line down. Prior to V39, the value of BitMap->BytesPerRow always happened to be the width of a bitmap scan line rounded up to the nearest multiple of 16, then divided by 8. And all BitMaps, whether allocated via AllocRaster, or AllocMem using the RASSIZE macro, or via OpenScreen, had this same BytesPerRow rounding. This rounding aligned every scanline on a word boundary to allow fetching by the word-oriented custom chips.

In addition, the IFF ILBM spec states that the scan lines of an ILBM BODY must be saved as width rounded up to the nearest multiple of 16 pixels (i.e., as an even number of bytes width). So it is not surprising that assumptions about BitMap->BytesPerRow crept into ILBM handling code.

ILBM BODY scan lines must still be stored as pixel width rounded up to a multiple of 16. But under V39 and AA, the higher bandwidth required to support new graphics modes requires that the scan lines of a displayable BitMap be aligned on larger boundaries. Therefore under V39 and AA, the BytesPerRow of a BitMap must not be confused with the correct storage width of an ILBM.

In addition, graphic applications must be very careful not to assume that all BitMaps of the same width will have the same BytesPerRow. >From V39 onwards, BitMaps allocated by different methods (e.g., OpenScreen(), AllocRaster(), AllocMem(), AllocBitMap()), or allocated for display by different chipsets (ECS, AA, etc.) or for different display modes or a promoted display mode, may have different scanline alignment restrictions and therefore a different BytesPerRow. Do not assume that bitplanes allocated by different methods can be swapped, or processor copied across scanline boundaries.

To test for BitMap->BytesPerRow problems, you generally must test on an AA machine with either mode promotion or explicit use of higher bandwidth (greater alignment restriction) modes.

Common symptoms of BitMap->BytesPerRow problems are

1. a skewing of the display, where the pixels on each successive scan line all appear shifted to either the left or right, creating a diagonal effect, or
2. excess blank space at the right edge of an ILBM or a display.

Watch out for interleaved bitmaps

If your application supports capturing any screen, you must watch out for the new interleaved bitmaps. An interleaved BitMap's BytesPerRow field is still the modulo for getting from any one pixel to the pixel directly below it, *but* it is no longer even related to the rounded up width of the screen or viewport. Instead, it is a *much* larger value which is actually the rounded up BitMap scan line width *times* the depth. Do not assume that BytesPerRow is related to the width of the display.

Note: The 2.0 Native Developer Update release of the NewIFF code had 2 major bugs. The screen.c module had a 1.3 incompatibility, and the ilbm.c module could not properly save an interleaved bitmap (such as the V39 Workbench screen). See the newer version 37.10 of the NewIFF code. This has been placed in our listings area on BIX, and sent to ADSP and Fred Fish. For full AA color support, see the NewIFF39 code (available to developers via BIX, ADSP, CIX and DevCon disks, and planned to be provided on the 3.0 Native Developer Update and given to Fred Fish).

Under V39, an interleaved bitmap can be detected by:

```
if (GetBitMapAttr(bitmap_ptr, BMA_FLAGS) & BMF_INTERLEAVED)
    printf("is interleaved");
```

Proper Printing of New Display Modes

When dumping a RastPort to printer under V36 and higher, the following IORequest field must contain a 32-bit modeid such as that returned by GetVPMODEID(viewport). You may want to allow the user the ability to print a display with a different modeid than it is being displayed in. Passing the full modeid allows the printer.device to properly control the aspect of the output, as long as the mode is available.

```

        ULONG    io_Modes;                                /* graphics viewport modes */

----- getcamg -----
From: /* ilbmr.c --- ILBM loading routines for use with iffparse */

/*
 * Returns CAMG or computed mode for storage in ilbm->camg
 *
 * ilbm->Bmhd structure must be initialized prior to this call.
 */
ULONG getcamg(struct ILBMInfo *ilbm)
{
    struct IFFHandle *iff;
    struct StoredProperty *sp;
    UWORD  wide,high,deep;
    ULONG modeid = 0L;

    if(! (iff=ilbm->ParseInfo.iff)) return(0L);

    wide = ilbm->Bmhd.pageWidth;
    high = ilbm->Bmhd.pageHeight;
    deep = ilbm->Bmhd.nPlanes;

    D(bug("Getting CAMG for w=%ld h=%ld d=%ld ILBM",wide,high,deep));

    /*
     * Grab CAMG's idea of the viewmodes.
     */
    if (sp = FindProp (iff, ID_ILBM, ID_CAMG))
    {
        modeid = (* (ULONG *) sp->sp_Data);

        /* knock bad bits out of old-style 16-bit viewmode CAMGs
         */
        if(! (modeid & MONITOR_ID_MASK)) ||
        ((modeid & EXTENDED_MODE) && !(modeid & 0xFFFF0000)))
            modeid &=
            (~ (EXTENDED_MODE | SPRITES | GENLOCK_AUDIO | GENLOCK_VIDEO | VP_HIDE));

        /* check for bogus CAMG like DPaintII brushes
         * with junk in upper word and extended bit
         * not set in lower word.
         */
        if((modeid & 0xFFFF0000) && !(modeid & 0x00001000)) sp=NULL;
    }

    if(! sp) {
        /*
         * No CAMG (or bad CAMG) present; use computed modes.
         */
        modeid = 0L;
        if (wide >= 640)            modeid = HIRES;
        if (high >= 400)           modeid |= LACE;

        /* This 6 planes == HAM or HALFBRITE is not
         * necessarily true anymore, but hopefully
         * all NEW programs are writing a proper CAMG chunk!!
         */
        if (deep == 6)
        {
            modeid |= ilbm->EHB ? EXTRA_HALFBRITE : HAM;
        }

        D(bug("No CAMG found - using mode $%08lx",modeid));
    }
}

```

```

    }

D(bug("getcamg: modeid = %#08lx",modeid));
return(modeid);
}

----- getaspect -----

bmhd->xAspect = 0; /* So we can tell when we've got it */
if(GfxBase->lib_Version >=36)
{
    if(GetDisplayInfoData(NULL, (UBYTE *)&DI,
        sizeof(struct DisplayInfo), DTAG_DISP, modeid))
    {
        bmhd->xAspect = DI.Resolution.x;
        bmhd->yAspect = DI.Resolution.y;
    }
}

/* If running under 1.3 or GetDisplayInfoData failed, use old method
 * of guessing aspect ratio
 */
if(! bmhd->xAspect)
{
    bmhd->xAspect = 44;
    bmhd->yAspect =
        ((struct GfxBase *)GfxBase)->DisplayFlags & PAL ? 44 : 52;
    if(modeid & HIRES)    bmhd->xAspect = bmhd->xAspect >> 1;
    if(modeid & LACE)     bmhd->yAspect = bmhd->yAspect >> 1;
}

```

NewIFF39: IFF Modules with AA Support

The NewIFF39 code modules and examples based on `iffparse.library` are designed as replacements for the original EA IFF code. The object code modules (with source provided) contain many high-level support functions for reading and writing IFF files and clipboard data, and for loading, saving, and displaying ILBM files in a 1.3 through 3.0/AA compatible manner. In some modules, it has been possible to retain much of the original EA IFF code. However, most structures and most higher level function interfaces have changed.

On the plus side, these new modules contain many easy-to-use functions for querying, loading, displaying, and saving ILBMs. Modules similar to these have been used in-house at Commodore during the development of the Display program and several other ILBM applications. The `screen.c` module provides powerful display-opening functions which are 1.3-compatible yet provide a host of new options under 2.0 and 3.0 such as centered overscan screens, full-video display clips, border transparency control, and autoscroll. These 39.x releases of the NewIFF code also support V39 and the AA chipset for full 8-bit-per-gun color and AA modes. Modules are provided for printing (`screendump`) and for preserving or adding chunks (`copychunks`). And the 8SVX example now actually plays samples and instruments. In addition, clipboard support is automatic for all applications that use the IFFP modules because `parse.c`'s `openfile()` interprets the filename `-c[n]` (i.e., `"-c"`, `"-c1"`, `"-c2"`, etc.) as clipboard unit `n`.

All of the applications with the NewIFF code require `iffparse.library` which has been part of the OS since Workbench 2.0. Please note that the 2.04 Workbench version of `iffparse.library` is a 1.3-compatible library, and that all of the modules and examples (with the exception of `Save8`) have been designed to take advantage of 2.0/3.0, but also work under 1.3. Developers who wish to distribute `iffparse.library` on their commercial products may execute a 2.0 Workbench license, or may get an amendment to their 1.3 Workbench license to allow distribution of `iffparse.library`.

The NewIFF39 modules contain enhanced code in many areas to take advantage of 2.0 and 3.0 features while remaining functional under 1.3 (with 2.04 `iffparse.library`). In addition, great effort has been put into identifying and replacing code sections which were inflexible or not upwards compatible. The code implements 32-bit CAMG, display database aspect ratios, 8-bit CMAP from 4, 8, or 32-bit color data, mode fallback when an ILBM's display mode is not available, overscan centering (except under 1.3) based on the user's closest Preference setting with display clips properly constrained to maximum limits, setting of as many colors as the destination can handle, default detection and adjustment of old 4-bit left-shifted CMAP values, and built-in support for up to 24 planes of data plus a Mask plane. Note that currently, the 24 planes plus 1 mask plane limit is hardcoded, but we expect to make this more flexible in a future release of the code, perhaps also adding alpha channel support.

Through the use of tag-like arrays of desired chunk IDs, applications can control which chunks are gathered by the parsing module. It is also possible to clone chunks for rewriting and for applications to add chunks to written FORMs. Applications may also pass additional screen tags to the display-opening module (screen.c) and may control much of this module's default behavior through flags (see `iffp/ilbmapp.h` of NewIFF39).

Most of the high-level function pairs provided in these modules have been designed to provide safe cleanup for themselves. For example, a `loadbrush()` that succeeds or fails at any point can be cleaned up via `unloadbrush`. The cleanup routines null out the appropriate pointers so that allocations will not be freed twice.

All applications use the `parse.c` module. The basic steps for using the `parse.c` module are:

- ☐ Define tag-like arrays of your desired chunks (readers only)
- ☐ Allocate one or more `[form]Info` structures as defined in `iffp/[form]app.h` (for example an `ILBMInfo` defined in `iffp/ilbmapp.h`).
- ☐ Initialize the `ParseInfo` part of these structures to the desired chunk arrays, and to an `IFFHandle` allocated via `iffparse AllocIFF()`.
- ☐ Use the provided high level load/save functions, or use the lower level `parse.c` `openfile()`, reader-only `parsefile()`/`getcontext()/nextcontext()`, and `closefile()`. The filename `-c[n]` may be used to read/write clipboard unit `n`.
- ☐ Clean up, `FreeIFF()`, and deallocate `[form]Infos`.

V39/AA Support

For NewIFF39, the `ILBMInfo` was extended to support a 32-bit-per-gun representation of the CMAP for use with the V39 color setting functions.

```
/* --- New --- */
WORD *colorrecord;      /* Passed to LoadRGB32 (ncolors,firstreg,table) */
Color32 *colortable32; /* 32-bit-per-gun representation of colors */
ULONG crecsize;         /* Bytes allocated including colorrecord WORDs */
```

For compatibility, the old style `WORD` array `colortable` of xRGB 4-bit values is still created for all loaded ILBMs. But the NewIFF39 code will also automatically allocate and create the 32-bit color table under V39 or higher, unless the calling application asks it not to by setting `IFFPB_NOCOLOR32` in `ILBMInfo->IFFPFlags`:

```
/* Don't allocate or use a 32-bit-per-gun Color Table under V39 or above */
#define IFFPB_NOCOLOR32 0
#define IFFPF_NOCOLOR32 (1L << IFFPB_NOCOLOR32)
```

By default, the NewIFF39 code will examine the low nibbles of an ILBM.CMAP to determine if it is an old-style left-shifted 4-bit-per-gun CMAP, and if all of the low nibbles of all usable registers are zero, the code will assume a 4-bit CMAP and scale the 4-bit values properly to 32 bits. This CMAP examination is disabled if either the ILBM.BMHD->Flags contains the flags BMHDF_CMAPOK (1L << 7), or if the calling application passes the IFFPF_CMAPOK flag in ILBMInfo->IFFPFlags. In either of these cases, the 8-bit CMAP values will be accepted as-is, and scaled to 32 bits.

Since V37, the NewIFF code has supported saving of 8-bit-significant CMAP data, and the function interface will accept 4-bit, 8-bit, or 32-bit per gun color data.

Currently, the NewIFF code does not make use of V39 Datatypes, and therefore does not have the capability to load non-IFF file formats. You may wish to link with the NewIFF modules but also add conditional application code to make use of Datatypes when datatypes.library is present. This would allow your program to be backwards compatible while taking even greater advantage of V39.

Locale Support

The NewIFF39 code does not use locale.library as-is. However, the code is prepared for localization. A catalog file of all module strings is provided (iffp.cd), and the module include file iffpstrings.h is generated from this iffp.cd file using CatComp 39.x. All string handling for the modules has been centralized in modules/iffpstrings.c which includes comments regarding locale support. See the Locale.Readme in the NewIFF39 archive for additional tips on writing localized IFF applications.

Important Notes

☐ Clipboard and File Handles

Most of the higher-level load functions keep the IFFHandle (file or clipboard) open. While the handle is open, you may use parse.c functions (such as findpropdata) *or* direct iffparse functions (FindCollection(), FindProp(),) for accessing the gathered chunks. However, it is not a good idea to keep a filehandle *or* the clipboard open. While a clipboard unit is open, no other applications can clip to the unit. And while a file is open, you can't write the file back out. So, instead of keeping the file or unit open, you can use copychunks (in copychunks.c) to create a copy of your gathered chunks, and do an early closefile() (parse.c). Then access and later write back out (if you wish) and deallocate your copied chunks via the routines in the copychunks module (findchunk, writechunklist, freechunklist).

General IFFParse Support Module

parse.c File/clipboard I/O and general parsing - openfile, closefile, parsefile, getcontext, nextcontext, contextis, currentchunkis, PutCk chunk writing function, and IFFerr text error routine

iffpstrings.c Centralized string and message handling for modules.

ILBM Read Modules

loadilbm.c High level ILBM load routines which are passed filenames (calls getbitmap) - loadbrush/unloadbrush, loadilbm/unloadilbm, and queryilbm

getbitmap.c brush/bitmap loading (non-display, calls ilbmr.c) - createbrush/deletebrush, getbitmap/freebitmap

getdisplay.c bitmap load/display (calls screen.c, ilbmr.c) - showilbm/unshowilbm, createdisplay/deletedisplay

screen.c 1.3/2.0/3.0 AA/ECS/non-ECS compatible screen/window module - opendisplay, openidscreen, modefallback, clipit

ilbmr.c Lower level ILBM body/color load routines (calls unpacker.c) - loadbody, loadcmap, getcolors/freecolors, alloccolortable, getcamg (gets or creates modeid)

unpacker.c BODY unpacker

ILBM Write Modules

saveilbm.c High level ILBM saving routines which are passed filenames (calls ilbmw.c) - screensave and saveilbm

ilbmw.c Lower level ILBM body/color save routines (calls packer.c) - InitBMHD, PutCMAP, PutBODY

packer.c BODY packer

Extra Modules

copychunks.c Chunk cloning and chunk list writing routines - copychunks, findchunk, writechunklist, freechunklist

screendump.c Screen printing module (iffparse not required)

bmprintc.c Module to output ILBM as C code

Include Files

iffp/#?.h This subdirectory may be kept in your current directory or in your main include directory.

Thanks to Steve Walton for his code changes for Manx/SAS compatibility, and to Bill Barton and John Bittner for their comments and suggestions.



IFF FORM AND CHUNK REGISTRY

The following is an alphabetical list of registered FORMs, generic chunks (shown as (any).chunkname), and registered new chunks for existing FORMs (shown as formname.chunkname). The center column describes where additional information on the FORM or chunk may be found. Items marked "EA_IFF" are described in the main chapters of the EA IFF specs. Those marked "IFF_TP" are described in the third-party specifications section. Items marked "propos" are proposals which have been submitted to CATS, some of which are private. And items marked with "-----" are private or yet unreleased specifications. New chunks and FORMS should be registered with CATS US, IFF Registry, 1200 Wilson Drive, West Chester, PA. 19380. Please make all submissions on Amiga diskette and include your address, phone, fax.

(any).ANNO	EA_IFF	EA IFF 85 Generic Annotation chunk
(any).AUTH	EA_IFF	EA IFF 85 Generic Author chunk
(any).CHRS	EA_IFF	EA IFF 85 Generic character string chunk
(any).CSET.doc	IFF_TP	chunk for specifying character set
(any).FRED	---	Private ASDG global chunk.
(any).FVER.doc	IFF_TP	chunk for 2.0 VERSION string of an IFF file
(any).HLID.doc	IFF_TP	HotLink IDentification (Soft-Logik)
(any).INFO.proposa	propos	This chunk contains data usually found in a file's .info file.
(any).JUNK.doc	IFF_TP	Always ignore this chunk.
(any).NAME	EA_IFF	EA IFF 85 Generic Name of art, music, etc. chunk
(any).TEXT	EA_IFF	EA IFF 85 Generic unformatted ASCII text chunk
(any).(c)	EA_IFF	EA IFF 85 Generic Copyright text chunk
8SVX	EA_IFF	EA IFF 85 8-bit sound sample form
8SVX.CHAN.PAN.doc	IFF_TP	Stereo chunks for 8SVX form
8SVX.SEQN.FADE.doc	IFF_TP	Looping chunks for 8SVX form
ACBM.doc	IFF_TP	Amiga Contiguous Bitmap form
AHAM	---	unregistered (???)
AIFF.doc	IFF_TP	Audio 1-32 bit samples (Mac,AppleII,Synthia Pro)
ANBM.doc	IFF_TP	Animated bitmap form (Framer, Deluxe Video)
ANIM.brush.doc	IFF_TP	ANIM brush format
ANIM.doc	IFF_TP	Cel animation form
ANIM.op6.doc	IFF_TP	Stereo (3D) Animations
ANIM.op7	---	unregistered (???)
ARC.proposal	propos	archive format proposal (old)
ARES	---	unregistered (???)
ATXT	---	temporarily reserved
AVCF.doc	IFF_TP	AmigaVision Course File
AVCO.doc	IFF_TP	AmigaVision Command (Nested in AVCF)
AVEV.doc	IFF_TP	AmigaVision EVent (Nested in AVCF)
BANK	---	Soundquest Editor/Librarian MIDI Sysex dump
BBSD	---	BBS Database, F.Patnaude,Jr., Phalanx Software
C100	---	Cloanto Italia private format
CAT	EA_IFF	EA IFF 85 group identifier
CHBM	---	Chunky bitmap (name reserved by Eric Lavitaky)
CLIP	---	CAT CLIP to hold various formats in clipboard
CMUS.proposal	propos	Common MUtical Score

CPFM	---	Cloanto Personal FontMaker (doc in their manual)
DCCL	---	DCCL - DCTV paint clip
DCPA	---	DCPA - DCTV paint palette
DCTV	---	DCTV - DCTV raw picture file
DECK	---	private format for Inovatronics CanDo
DEEP.doc	IFF_TP	Chunky pixel image files (Used in TV Paint)
DR2D.doc	IFF_TP	2-D Object standard format
DRAW	---	reserved by Jim Bayless, 12/90
DTYP.doc	IFF_TP	DataTypes Identification
EXEC.proposal	propos	Proposed FORM for executable (loadseg-able) code.
FANT.doc	IFF_TP	Fantavision movie format
FAX3	---	private GPSoftware FAX format, no longer used.
FAXX.GPHD.doc	IFF_TP	Additional header info for FAXX FORMs
FAXX.doc	IFF_TP	FAXX (Facsimile image FORM)
FIGR	---	Deluxe Video - reserved
FILM	---	LIST FILM - For storing ILBM's with interleaved 8SVX audio
FNTR	EA_IFF	EA IFF 85 reserved for raster font
FNTV	EA_IFF	EA IFF 85 reserved for vector font
FORM	EA_IFF	EA IFF 85 group identifier
FTXT	EA_IFF	EA IFF 85 formatted text form
GRYP.proposal	propos	byteplane storage proposal (copyrighted)
GSCR	EA_IFF	EA IFF 85 reserved gen. music score
GUL.proposal	propos	user interface storage proposal (private)
HEAD.doc	IFF_TP	Flow - New Horizons Software
ILBM	EA_IFF	EA IFF 85 raster bitmap form
ILBM.3DCM	---	reserved by Haitex
ILBM.3DPA	---	reserved by Haitex
ILBM.ASDG	---	private ASDG application chunk
ILBM.BHBA	---	private Photon Paint chunk (brushes)
ILBM.BHCP	---	private Photon Paint chunk (screens)
ILBM.BHSM	---	private Photon Paint chunk
ILBM.CLUT.doc	IFF_TP	Color Lookup Table chunk
ILBM.CMYK.doc	IFF_TP	Cyan, Magenta, Yellow, & Black color map (Soft-Logik)
ILBM.CNAM.doc	IFF_TP	Color naming chunk (Soft-Logik)
ILBM.CTBL.DYCP.doc	IFF_TP	Newtek Dynamic Ham color chunks
ILBM.DCTV	---	reserved
ILBM.DGVW	---	private Newtek DigiView chunk
ILBM.DPL.doc	IFF_TP	Dots per inch chunk
ILBM.DPPV.doc	IFF_TP	DPaint perspective chunk (EA)
ILBM.DRNG.doc	IFF_TP	DPaint IV enhanced color cycle chunk (EA)
ILBM.EPSF.doc	IFF_TP	Encapsulated Postscript chunk
ILBM.PCHG.doc	IFF_TP	Line by line palette control information (Sebastiano Vigna)
ILBM.PRvw.proposal	propos	A mini duplicate ILBM used for preview (Gary Bonham)
ILBM.TMAP	---	Transparency map (temporarily reserved)
ILBM.VTAG.proposal	propos	Viewmode tags chunk suggestion
ILBM.XBML.doc	IFF_TP	eXtended BitMap Information (Soft-Logik)
ILBM.XSSL.doc	IFF_TP	Identifier chunk for 3d X-Specs image. (Haitex)
IOBJ	---	reserved by Seven Seas Software
IODK	---	reserved for Jean-Marc Porchet at Merging Technologies
ITRF	---	reserved
JMOV	---	Reserved for Merging Technologies
LIST	EA_IFF	EA IFF 85 group identifier
MFAX	---	Reserved for TKR GmbH & Co.
MIDI	---	Circum Design

MOVI	---	LIST MOVI - private format
MSCX	---	private Music-X format
MSMP	---	temporarily reserved
MTRX.doc	IFF_TP	Numerical data storage (MathVision - Seven Seas)
NSEQ	---	Numerical sequence (Stockhausen GmbH)
OB3D.proposal	propos	Proposal for a stadard 3D object format
OCMP	EA_IFF	EA IFF 85 reserved computer prop
OCPU	EA_IFF	EA IFF 85 reserved processor prop
OPGM	EA_IFF	EA IFF 85 reserved program prop
OSN	EA_IFF	EA IFF 85 reserved serial num prop
PGBT.doc	IFF_TP	Program traceback (SAS Institute)
PICS	EA_IFF	EA IFF 85 reserved Macintosh picture
PLBM	EA_IFF	EA IFF 85 reserved obsolete name
PMBC.proposal	propos	Reserved for Black Belt Systems 91.12.01
PREF	---	Reserved by Commodore for user preferences data, currently private.
PROP	EA_IFF	EA IFF 85 group identifier
PRSP.doc	IFF_TP	DPaint IV perspective move form (EA)
PTCH	---	Patch file format (SAS Institute)
PTXT	---	temporarily reserved
RGB4	---	4-bit RGB (format not available)
RGBN-RGB8.doc	IFF_TP	RGB image forms, Turbo Silver (Impulse)
RGBX	---	temporarily reserved
ROXN	---	private animation form
SAMP.doc	IFF_TP	Sampled sound format
SC3D	---	private scene format (Sculpt-3D)
SHAK	---	private Shakespeare format
SHO1	---	Reserved by Gary Bonham (private)
SHOW	---	Reserved by Gary Bonham (private)
SMUS	EA_IFF	EA IFF 85 simple music score form
SPLT.doc	IFF_TP	ASDG's File SPLiTing system
SSRE	---	Reserved for Merging Technologies 92.05.04
SWRT	---	unregistered (???)
SYTH	---	SoundQuest Master Librarian MIDI System driver
TCDE	---	reserved by Merging Technologies
TDDD.doc	IFF_TP	3-D rendering data, Turbo Silver (Impulse)
TERM	---	unregistered (???)
TREE.doc	IFF_TP	Storage of arbitrary data structures as trees (or nested lists).
TRKR.proposal	propos	TRacKeR style music module format proposal
UNAM	EA_IFF	EA IFF 85 reserved user name prop
USCR	EA_IFF	EA IFF 85 reserved Uhuru score
UVOX	EA_IFF	EA IFF 85 reserved Uhuru Mac voice
VIDEO	---	private Deluxe Video format
WORD.doc	IFF_TP	ProWrite document format (New Horizons)
YUVN.doc	IFF_TP	For storage of Y:U:V image data (MacroSystem)





3.0 AmigaGuide™

by David N. Junod

Introduction

The standard Amiga keyboard sports a HELP key, yet there has been no system provided support for this key. Now, there is AmigaGuide, which provides a standard method of displaying help and other on-line documentation to the user.

Sections marked with ** indicate that the option is only available in pre-3.0 versions of AmigaGuide.

Capabilities

AmigaGuide uses an Intuition window that contains a scroll bar, buttons and pull-down menus, to display plain ASCII text files or AmigaGuide databases.

An AmigaGuide database is a set of related documents contained in one file. Each document may contain references to other documents, using what is called a link. A document may contain any number of links, pointing to any number of other documents. When the user selects a link, the document that the link points to will be displayed. The user may then use the links to read through the database, following whatever path he may choose. The technical term for AmigaGuide's abilities is hypertext.

The user may at any time print a document or a portion of the document. **He may also send portions of a document to the clipboard, for use in other applications.

Using ARexx, the user may write scripts, or an application could provide scripts, to control AmigaGuide.

**Cross-reference tables can be loaded that specify where a keyword, or phrase is defined. The user can then use AmigaGuide's Find Document facility to quickly display a document based on keyword, without having to know the name of the database that it is located in.

AmigaGuide provides a unique feature to hypertext systems, called Dynamic Nodes. A Dynamic Node is a hypertext or plain text document that is generated in real-time as opposed to coming from a static file. An application that generates Dynamic Nodes is called a Dynamic Node Host.

Interfacing

AmigaGuide databases are accessed in three different ways:

- ☐ Databases can be browsed directly from the Workbench or Shell using the MultiView utility.
- ☐ AmigaGuide support can be added to an existing application that supports ARexx by using AmigaGuide's ARexx function host capabilities.
- ☐ Applications can use the functions of AmigaGuide to provide help on gadgets, menus and windows. For example, the user could position the pointer over any gadget or menu item, press help, and the appropriate document would be displayed in the AmigaGuide window. The application could also have AmigaGuide display a pertinent portion of the current project.

Other Uses

In addition to help or on-line documentation, AmigaGuide has other possible uses.

Tutorials

An application that has an ARexx port and supports AmigaGuide could set up a help system that not only provides help, but also gives examples. The user could read about a feature, then click on the EXAMPLE button, which would run an ARexx script that would give an example of use. For instance, to show Pattern Fill, the script could draw a circle, select a pattern, and then fill the circle.

Computer Aided Instruction

The student could read about different topics, following links. A multiple choice quiz could be set up at the end where the questions and answers run ARexx scripts to accumulate the score.

Program by Query

Many programmers develop using a Cut & Paste technique. They clip modules from various applications or utilities they have written and paste them together to build new applications. A database of these different code fragments could be set up (such as loading and saving ILBMs, playing sounds, etc.) and you could step through, answering questions, while the sections you need are being appended to a new source file.

USER PREFERENCES

AmigaGuide allows a number of items to be tailored to the users' preference. These preference items are stored in environment variables. The AmigaDOS command SetEnv can be used to set any of these variables.

In order to set any of the following environment variables, an ENV:AmigaGuide directory must be made.

```
makedir ENV:AmigaGuide
```

A Preferences Editor that sets AmigaGuide preferences would write in the ENV:AmigaGuide directory when "Use" is selected, and write in the ENVARC:AmigaGuide directory when "Save" is selected.

Following is a list of the variable names, and what they control.

Path

This variable contains the list of directory names that AmigaGuide will search through when it attempts to open a database. The directory names are separated by a space.

```
SetEnv AmigaGuide/Path "Workbench:Autodocs Workbench:Includes"
```

**Pens

This variable provides the user with the ability to specify the colors to use for the various renderings that AmigaGuide performs.

```
SetEnv AmigaGuide/Pens <abcdefgh>
```

Where:

a = Background pen	b = Button text pen
c = Button background pen	d = Highlighted button text pen
e = Highlighted button background pen	f = Outline pen
g = Highlight outline pen	h = Text on background pen

```
SetEnv AmigaGuide/Pens 21213001
```

Internally, AmigaGuide subtracts "0" from the pen number, so values can range from 0 to 207.

****Text**

Used to specify the graphical style that the links are presented in. The possible styles are:

- BUTTON** Draw a raised border around the text (default).
- UNDERLINE** Underline the text.
- BOLD** Bold the text.
- ITALIC** Italicize the text.

SetEnv AmigaGuide/Text **BUTTON**

Authoring AmigaGuide Documents

Authoring an AmigaGuide database, or any hypertext database for that matter, is a difficult task. It takes a lot of insight into the subject matter and how the pieces relate to each other. A database must consist of documents that are related. Documents must be broken into manageable chunks, and links carefully thought out. A document should consist of information dealing with one topic and should contain links to other related documents.

An AmigaGuide database is ASCII text with embedded commands that tell AmigaGuide how to interpret the database. A database should consist of a main table of contents and a number of related documents.

Label Commands

These are commands that can be used within a database. Commands must start in the first column of a line. If a line begins with an @ sign, then it is interpreted as a command.

@DATABASE <name>

Must be the very first line of an AmigaGuide document.

@MASTER <path>

Complete path of the source document used to define this AmigaGuide database.

@AUTHOR <name>

The author of the database.

@(C) <copyright>

The copyright notice for the database.

@\$VER: <AmigaDOS version string>

Specify the version of the database. This command must always be uppercase.

@FONT <name> <size>

The font to use for the database.

@INDEX <name/node>

The name of the index node, which will be accessed by the "Index" button.
Can be a node in an external database.

@HELP <name/node>

The name of the help node, which will be accessed by the "Help" button. Can be a node in an external database.

@NODE <name> <title>

Indicate the start of a node (page/article/section). The first node, or main node, must be named MAIN. MAIN must be the master table of contents for the database.

@DNODE <name>

Indicates the start of a dynamic node. The AmigaGuide system uses the callback hooks to obtain the document from a document provider.

@WIDTH <chars>

How wide, in characters, the largest document is.

@HEIGHT <chars>

How high, in characters, the largest document is.

<remark> @REMARK <remark>

Remark (not displayed to the user).

Node Label Commands

These are commands that can be used within an @NODE.

@ENDNODE <name>

Indicate the end of a node.

@TITLE <title>

Title to display in the title bar of the window during the display of this node.

@TOC <node name>

Name of the node that contains the table of contents for this node. Defaults to MAIN. This is the node that is displayed when the user presses the "Contents" button.

@PREV <node name>

Node to display when the user selects "< Browse"

@NEXT <node name>

Node to display when the user selects "Browse >"

@FONT <name> <size>

Specify the font to use for the node.

@{<label> <command>}

Indicate a textual link point. Can be anywhere in a line. Starting with 3.0, AmigaGuide can link to graphics, sounds, animations and other DataTypes.

\@

A backslash in front of the @ sign is used to escape it.

Attributes

The following list of attributes can be applied to the text within a node. This feature is only supported with the 3.0 version of AmigaGuide.

@{B}

Turn bold on.

@{UB}

Turn bold off.

@{I}

Turn italic on.

@{UI}

Turn italic off.

@{U}

Turn underline on.

@{UU}

Turn underline off.

@{FG <color>}

Change the foreground text color. Color can be:

Text	Shine
Shadow	Fill
FillText	Background
Highlight	

@{bg <color>}

Change the background color. The same colors can be used as in the FG command.

Action Commands

These are commands that can be assigned to a link point.

****ALINK <name> <line>**

Load the named node into a new window, with <line> at the top of the display.

BEEP

Cause a display beep in the screen that the AmigaGuide window resides in.

CLOSE

Close the window (should only be used on windows that were started with alink).

LINK <name> <line>

Load the named node, with <line> at the top of the display.

RX <command>

Execute an ARexx macro.

RXS <command>

Execute an ARexx string file. To display a picture, use 'ADDRESS COMMAND DISPLAY <picture name>', to display a text file 'ADDRESS COMMAND MORE <doc>'. Note that with 3.0 it is possible to display text, graphics or sounds within the AmigaGuide window.

SYSTEM <command>

Execute an AmigaDOS command.

QUIT

Shutdown the current database.

Example AmigaGuide Database

The following is an example of an AmigaGuide database. It doesn't contain any 'useful' information, but it does show the usage of some of the commands.

```
@database "example.guide"
@master "example.doc"

@node Main "Example AmigaGuide database"

Table of Contents
@("ARexx" link ARexx)
@("Shell" link Shell)
@("Workbench" link Workbench)
@endnode
```

```

@node ARExx
Put something here about @({b)ARExx@{ub)}.
@endnode

@node Shell
Put something here about the @({b)Shell@{ub)}.
@endnode

@node Workbench
Put something here about @({b)Workbench@{ub)}. Say that it has @{"icons" link icon}.
@endnode

@icon "Workbench Icons"
Those little pictures that you can drag around.
@endnode

```

AREXX SCRIPTS

It is possible to control AmigaGuide using ARExx. Each occurrence of AmigaGuide has an ARExx port. The AmigaGuide shared system library is also an ARExx function host.

Port Naming

The default port name is AMIGAGUIDE.# where # is the occurrence. With the ****AmigaGuide** utility, a port name can be specified as a command line argument. An application with an AmigaGuide interface can also provide the port name.

ARExx Commands

Any of the following action commands are also ARExx commands. All commands are not case-sensitive.

****ALINK <name> <line>**

Load the named node into a new window, with <line> at the top of the display.

BEEP

Cause a display beep in the screen that the AmigaGuide window resides in.

CLOSE

Close the window (should only be used on windows that were started with alink).

LINK <name> <line>

Load the named node, with <line> at the top of the display.

SYSTEM <command>

Execute an AmigaDOS command.

QUIT

Shutdown the current database.

ARexx Functions

The amigaguide.library is an ARexx function library. The library can be added as a function host with the following lines:

```
/* Load the AmigaGuide library as a function host */  
IF ~SHOW('L','amigaguide.library') THEN  
    CALL ADDLIB('amigaguide.library',0,-30)
```

It supports the following functions (function names are not case-sensitive).

ShowNode PUBSCREEN/K,DATABASE/K,NODE/K,LINE/N

Display a node on the named screen. Defaults to the Main node on the Workbench screen. If DATABASE isn't specified, then will search through the cross-reference list to get the database name.

LoadXRef NAME/K

Load a cross-reference file into memory.

GetXRef NODE/K

Return information on NODE. Format of the text string returned is "NODE"
"DATABASE" TYPE LINE.

ExpungeXRef

Flush the cross-reference list from memory.

Adding an AmigaGuide Interface to Your Application

Applications can add AmigaGuide support using the functions within the amigaguide.library.

The AGHelp example on disk illustrates how to add simple context sensitive help to an application. It uses the new 3.0 Intuition gadget help routines to determine which gadget the pointer is over, and uses AmigaGuide to display the help text.

The AdvAGHelp example on disk shows a more complicated context sensitive help system. Like AGHelp, it uses the new 3.0 Intuition functions, but also shows:

Continuous Help

Display help information as the user moves the mouse around over the objects of the user interface.

Project Information

Using current project information or user interface state information (such as a gadget or menu being disabled) in your help documents.

Cross Reference Files

AmigaGuide allows cross-reference tables to be loaded that specify what document a keyword is defined in. **This cross-reference table is used by the "Find Document" requester to locate a node. It is also used by the AD2AG utility to construct hypertext versions of the system Autodoc files.

A cross-reference file follows a layout similar to the devs:mountlist format. The table itself starts with a line that consists of the keyword XREF: and ends with a line that contains a # as the only uncommented character. Comments can be included in C-style format, beginning with "/*" and ending with "*/".

```
/* This is a comment */
XREF:
...    "Gadget" "intuition/intuition.h"          215 3
...
#
```

A cross-reference entry consists of four words:

Keyword

The keyword that is being defined.

File

The ASCII file or database that the keyword is defined in.

Line

The line within the node that the keyword is defined on.

Type

This field indicates the type of keyword. Possible values are.

- | | |
|--------------------------------|--------------------------------|
| 0 Generic AmigaGuide link. | 1 Describes a function. |
| 2 Describes a command. | 3 Points to an include file. |
| 4 Describes a macro. | 5 Describes a structure. |
| 6 Describes a structure field. | 7 Describes a type definition. |
| 8 Describes a define. | |

Loading a Cross Reference List

A global cross-reference list can be loaded from disk using the LoadXRef() function. The format is.

```
LONG success;  
BPTR lock;  
STRPTR name;  
  
success = LoadXRef(lock, name);
```

The arguments are

lock

Lock on the directory where the file is located. May be NULL.

name

Name of the cross-reference file to load. LoadXRef will search the user preference path.

It returns

-1

Indicates that the load was aborted by a Ctrl-C.

0

Unable to load the file.

1

Successfully loaded the file.

2

No changes have been made since the last time that this file was loaded.

Access to the Cross Reference List

An application can use the GetAmigaGuideAttr() function to obtain a pointer to the cross-reference list. The application then may search through the list, or even save the list to disk. Note that access to this list is read-only, and must be enclosed between a call to LockAmigaGuideBase() and UnlockAmigaGuideBase().

```

struct List *list;
LONG key;

/* Lock the AmigaGuideBase for exclusive access */
key = LockAmigaGuideBase(NULL);

/* Get a pointer to the cross-reference list */
if (GetAmigaGuideAttr(AGA_XRefList, NULL, &list))
{
    /* Do something with the list */
}

/* Unlock AmigaGuideBase */
UnlockAmigaGuideBase(key);

```

A cross-reference list consists of nodes of struct XRef, defined in <libraries/amigaguide.h>.

```

/* Cross-reference node */
struct XRef
{
    struct Node xr_Node;    /* Embedded node */
    UWORD xr_Pad;          /* Padding */
    struct DocFile *xr_DF; /* (Private) Document defined in */
    STRPTR xr_File;        /* Name of document file */
    STRPTR xr_Name;        /* Name of item */
    LONG xr_Line;          /* Line defined at */
};

#define XRSIZE (sizeof (struct XRef))

```

Following are the field definitions.

xr_Node

Embedded node structure. `xr_Node.ln_Name` points to `xr_Name`.
`xr_Node.ln_Type` contains the type of the keyword.

xr_Pad

Used to align the remaining fields.

xr_DF

Private pointer.

xr_File

Pointer to the name of the file that `xr_Name` is defined in.

xr_Name

Pointer to the keyword.

xr_Line

The line, within `xr_File`, that `xr_Name` is defined on.

Dynamic Node Host

AmigaGuide provides a unique feature to hypertext systems called Dynamic Nodes. A Dynamic Node is a hypertext or plain text document that is generated in real-time as opposed to coming from a static file. An application that generates Dynamic Nodes is called a Dynamic Node Host.

If a link point within a document isn't resolved, it will query a list of Dynamic Node Hosts to see if any one of these external applications can resolve the node. This feature allows for dynamic interaction with constantly changing data. It is useful for AmigaGuide authoring tools, interactive development environments and extremely context sensitive help systems, to name a few.

Dynamic Nodes has been implemented using an Object Oriented Programming paradigm. When a link point hasn't been resolved an HM_FINDNODE message is sent to each Dynamic Node Host on the list. Once the node has been found, an HM_OPENNODE is sent to the Dynamic Node Host that the node belongs to. HM_CLOSENODE is sent to the host once the node is exited.

Initializing a Dynamic Node Host

In order for an application to register itself as a Dynamic Node Host, it must initialize a hook and add the hook to the AmigaGuide Dynamic Node list, using the AddAmigaGuideHost()

The hook structure as defined in <utility/hooks.h>.

```
/* Standard hook structure */
struct Hook {
    struct MinNode h_MinNode;
    ULONG (*h_Entry)(); /* assembler entry point */
    ULONG (*h_SubEntry)(); /* often HLL entry point */
    VOID *h_Data; /* owner specific */
};
```

The AddAmigaGuideHost() function returns a pointer to an AmigaGuideHost structure. This structure, defined in <libraries/amigaguide.h>, is as follows.

```
/* Callback handle */
struct AmigaGuideHost {
    struct Hook agh_Dispatcher; /* Dispatcher */
    ULONG agh_Reserved; /* Must be 0 */
    ULONG agh_Flags;
    ULONG agh_UseCnt; /* Number of open nodes */
    APTR agh_SystemData; /* Reserved for system use */
    APTR agh_UserData; /* Anything you want... */
};
```

Following are the field definitions for the AmigaGuideHost structure.

agh_Dispatcher

This is a copy of the Hook that was passed to AddAmigaGuideHost().

agh_UserData

Can be manipulated by the Dynamic Node Host any way it sees fit.

The other fields are not to be manipulated in any way.

Removing a Dynamic Node Host

A Dynamic Node Host is removed using the RemoveAmigaGuideHost() library function. The application must successfully remove the hook before exiting, otherwise AmigaGuide would end up calling the hook function, that has been unloaded from the system, causing a system crash.

The following code fragment illustrates how to initialize and remove a Dynamic Node Host.

```
#include <exec/types.h>
#include <libraries/amigaguide.h>
#include <clib/exec_protos.h>
#include <clib/amigaguide_protos.h>
#include <pragmas/exec_pragmas.h>
#include <pragmas/amigaguide_pragmas.h>
#include <stdio.h>

extern struct Library *SysBase, *DOSBase;
struct Library *AmigaGuideBase;

#define ASM __asm
#define REG(x) register __ ## x

ULONG __saveds dispatchDNH(struct Hook *, STRPTR, Msg);
ULONG ASM hookEntry(REG(a0) struct Hook *, REG(a2) VOID *, REG(a1) VOID *);

/* Callback hook dispatcher */
ULONG __asm hookEntry (
    REG(a0) struct Hook *h,
    REG(a2) VOID *obj,
    REG(a1) VOID *msg)
{
    /* Pass the parameters on the stack */
    return ((h->h_SubEntry)(h, obj, msg));
}

main (int argc, char **argv)
{
```

```

struct Hook hook;
AMIGAGUIDEHOST hh;

/* amigaguide.library works with 1.3 and newer versions of the OS */
if (AmigaGuideBase = OpenLibrary ("amigaguide.library", 33))
{
    /* Initialize the hook */
    hook.h_Entry = hookEntry;
    hook.h_SubEntry = dispatchDNH;

    /* Add the AmigaGuideHost to the system */
    if (hh = AddAmigaGuideHost (&hook, "ExampleHost", NULL))
    {
        printf("Added AmigaGuideHost 0x%lx", hh);

        /* Wait until we're told to quit */
        Wait (SIGBREAKF_CTRL_C);

        printf ("Remove AmigaGuideHost 0x%lx", hh);

        /* Try removing the host */
        while (RemoveAmigaGuideHost (hh, NULL) > 0)
        {
            /* Wait a while */
            printf (".");
            Delay (250);
        }
        printf ("");
    }
    else
    {
        printf ("Couldn't add AmigaGuideHost");
    }

    /* close the library */
    CloseLibrary (AmigaGuideBase);
}
}

```

Handling Dynamic Node Host Messages

Once the Dynamic Node Host has been added to AmigaGuide, it can start receiving messages for different requests.

Currently, AmigaGuide supports the following methods, or message types, for a Dynamic Node Host.

HM_FINDNODE

When AmigaGuide can't resolve a link, then it sends an HM_FINDNODE message to all Dynamic Node Hosts to see which host defines the node.

HM_OPENNODE

Once AmigaGuide locates the host that defines a node, using the HM_FINDNODE message, then the HM_OPENNODE message is sent to that host to ask it to open the node.

HM_CLOSENODE

Once the user has closed all occurrences of a Dynamic Node, then AmigaGuide sends the HM_CLOSENODE message to the host that opened the node.

HM_EXPUNGE

AmigaGuide sends this message to all Dynamic Node Hosts when the Expunge vector of amigaguide.library is invoked, or the ExpungeDataBases() function is called.

Several of the methods receive a TagItem array as an argument. Currently the following tags are supported. The tag values are defined in <libraries/amigaguide.h>.

HTNA_Screen

A pointer to the screen on which source AmigaGuide window resides.

HTNA_Pens

The pen array associated with the screen.

HTNA_Rectangle

A Rectangle structure (defined in <graphics/gfx.h>) containing the dimensions of the window.

HTNA_HelpGroup

A unique numeric identifier associated with an AmigaGuide client. This tag is new for 3.0.

Each method requires one or more parameters. The MethodID is the only common parameter for each method.

HM_FINDNODE

Used to locate the Dynamic Node Host that a node is defined by. When a Dynamic Node Host receives a HM_FINDNODE message for a node that it owns, it should reply with TRUE, otherwise it must respond with FALSE.

The HM_FINDNODE method receives the following arguments:

```

/* HM_FINDNODE */
struct opFindHost {
    ULONG MethodID;
    struct TagItem *ofh_Attrs; /* R: Additional attributes */
    STRPTR ofh_Node;          /* R: Name of node */
    STRPTR ofh_TOC;           /* W: Table of Contents */
    STRPTR ofh_Title;         /* W: Title to give to the node */
    STRPTR ofh_Next;          /* W: Next node to browse to */
    STRPTR ofh_Prev;          /* W: Previous node to browse to */
};

```

The field definitions are as follows

ofh_Attrs

This field contains a pointer to a TagItem array of attributes for the message.
This field is read-only.

ofh_Node

The name of the node to open. This field is read-only. It is possible for this name to contain parameters that need to be parsed. For example, the command that triggered the link could have been:

Link "snd/beep 320"

In which case, the ofh_Node field would contain:

beep 320

ofh_TOC

The Table of Contents to assign to this node. This is the name of the node to link to, if the "Contents" button is pressed. This field can be written to (not implemented).

ofh_Title

The title to assigned to this node. This field can be written to.

ofh_Next

The name of the logical next node. This is the name of the node to link to if the "Browse >" button is pressed. This field can be written to.

ofh_Prev

The name of the logical previous node. This is the name of the node to link to if the "< Browse" button is pressed. This field can be written to.

HM_OPENNODE

Once AmigaGuide locates the host that defines a node, using the HM_FINDNODE message, then the HM_OPENNODE message is sent to that host to ask it to open the node. If the Dynamic Node Host is able to open the node, then it should respond with TRUE, otherwise respond with FALSE.

The HM_OPENNODE method receives the following arguments:

```
/* HM_OPENNODE, HM_CLOSENODE */
struct opNodeIO {
    ULONG MethodID;
    struct TagItem *onm_Attrs; /* R: Additional attributes */
    STRPTR onm_Node; /* R: Node name and arguments */
    STRPTR onm_FileName; /* W: File name buffer */
    STRPTR onm_DocBuffer; /* W: Node buffer */
    ULONG onm_BuffLen; /* W: Size of buffer */
    ULONG onm_Flags; /* RW: Control flags */
};
```

The field definitions are as follows

onm_Attrs

This field contains a pointer to a TagItem array of attributes for the message.
This field is read-only.

onm_Node

The name of the node to open. This field is read-only. It is possible for this name to contain parameters that need to be parsed. For example, the command that triggered the link could have been:

Link "snd/beep 320"

In which case, the onm_Node field would contain:

beep 320

onm_FileName

If you want AmigaGuide to read a particular node from disk, then supply the file name here. The file can either be a straight ASCII file or an AmigaGuide document (not a database). The application can write to this field.

onm_DocBuffer

If you are dynamically creating a node in memory, then use this field to point to the buffer. If this field is used, then the onm_BuffLen field must be filled in also. The application is in charge of freeing onm_DocBuffer when it is done (indicated by a HM_CLOSENODE message). The application can write to this field.

onm_BuffLen

The length of the buffer that onm_DocBuffer points to. The application can write to this field.

onm_Flags

These are control flags that the Dynamic Node Host can set.

HTNF_KEEP

Don't flush this node from memory until the database is closed. This will delay the HM_CLOSENODE message until the database is closed.

HTNF_ASCII

The node is straight ASCII, doesn't contain any AmigaGuide keywords.

HTNF_CLEAN

Remove the node from the database as soon as it is closed.

HTNF_DONE

This flag is used to indicate to AmigaGuide that the Dynamic Node Host already took care of presenting the node, and that there is no need for AmigaGuide to present it. This is useful for playing sounds, animations, or even debugging information.

HM_CLOSENODE

Once the user has closed all occurrences of a Dynamic Node, then AmigaGuide sends the HM_CLOSENODE message to the host that opened the node. If the Dynamic Node Host is able to close the node, then respond with TRUE, otherwise respond with FALSE.

The HM_CLOSENODE message uses the same message structure as HM_OPENNODE.

HM_EXPUNGE

AmigaGuide sends this message to all Dynamic Node Hosts when the Expunge vector of amigaguide.library is invoked, or the ExpungeDataBases() function is called. The Dynamic Node Host should free as much memory as it possibly can.

The HM_EXPUNGE method receives the following arguments:

```
/* HM_EXPUNGE */
struct opExpungeNode {
    ULONG MethodID;
    struct TagItem *oen_Attrs; /* R: Additional attributes */
};
```

The field definitions are as follows

oen_Attrs

Currently, no attributes passed.

Message Dispatcher

The following is an example of Dynamic Node Host message dispatcher. Since this is executed with a callback hook, it is being run on the calling process' task, not the Dynamic Node Host process. Because of that, it is necessary to load the global data segment using `geta4()` or `__saveds`.

```
ULONG __saveds
dispatchAmigaGuideHost (struct Hook *h, STRPTR db, Msg msg)
{
    struct opNodeIO *onm = (struct opNodeIO *) msg;
    ULONG retval = 0;

    switch (msg->MethodID)
    {
        /* Does this node belong to you? */
        case HM_FINDNODE:
        {
            struct opFindHost *ofh = (struct opFindHost *) msg;

            DB (kprintf ("Find [%s] in %s", ofh->ofh_Node, db));

            /* See if they want to find our table of contents */
            if ((strcmp (ofh->ofh_Node, "main")) == 0)
            {
                /* Return TRUE to indicate that it's your node, otherwise return FALSE. */
                retval = TRUE;
            }
            else
            {
                Display (onm); /* Display the name of the node */

                /* Return TRUE to indicate that it's your node, otherwise return FALSE. */
                retval = FALSE;
            }
        }
        break;

        /* Open a node. */
        case HM_OPENNODE:
        {
            DB (kprintf ("Open [%s] in %s", ofh->onm_Node, db));
            /* See if they want to display our table of contents */
            if ((strcmp (onm->onm_Node, "main")) == 0)
            {
                /* Provide the contents of the node */
            }
        }
    }
}
```

```

        onm->onm_DocBuffer = TEMP_NODE;
        onm->onm_BuffLen = strlen (TEMP_NODE);
    }
    else
    {
        Display(onm);      /* Display the name of the node */

        /* Indicate that we want the node removed from our database, and that we handled
        * the display of the node
        */
        onm->onm_Flags |= (HTNF_CLEAN | HTNF_DONE);
    }

    /* Indicate that we were able to open the node */
    retval = TRUE;
    break;

    /* Close a node that has no users. */
    case HM_CLOSENODE:
        DB (kprintf("Close[%s] in %s", onm->onm_Node, db));

        /* Indicate that we were able to close the node */
        retval = TRUE;
        break;

    /* Free any extra memory */
    case HM_EXPUNGE:
        DB (kprintf ("Expunge [%s]", db));
        break;

    default:
        DB (kprintf {"Unknown method %ld",msg->MethodID});
        break;
    }
    return (retval);
}

```

Developer Specific Utilities

On the DevCon disks are a number of utilities for AmigaGuide that would be of special interest to the developer.

AD2AG

Scans Autodocs for function and command names, scans the INCLUDE: directory for .h include files, and scans the include files for structure definitions and typedefs. Also parses Autodoc files and constructs a corresponding AmigaGuide database. It resolves links to functions, commands, include files, structures and typedefs.

Glossary

Autodoc

Documentation extracted from source code.

Browse

Navigate sequentially through a series of documents, instead of via links.

Cross-reference table

A table that consists of the following information:

Keyword A word or phrase

Database Name of the database that the keyword is defined in.

Line The line that the keyword is defined on within the database.
This only applies if the database is a straight text file (such as an include file).

Type What type the keyword is, such as a "normal", function, command, include file, or structure.

Database

A file that consists of multiple documents.

Document

A block of text, constrained to one subject. Also called a node.

Dynamic Node

A Dynamic Node is a hypertext, or plain text, document that is generated in real-time, or from live data, as opposed to coming from a static file.

Dynamic Node Host

An application that generates Dynamic Nodes.

Link

A word, or phrase, within a document that is linked to another document.

Node

A block of text, constrained to one subject. Also called a document.

Retrace

To follow, in a reverse direction, the path taken through a series of documents.

Table of Contents

A list of documents, categorized by type.

Recommended Reading

Ben Shneiderman & Greg Kearsley, *Hypertext Hands-On!*, Addison-Wesley Publishing Company, ISBN 0-201-13546-9

Philip Seyer, *Understanding Hypertext Concepts and Applications*, Windcrest Books, ISBN 0-8306-9108-1 (hardcover) 0-8306-3308-1 (paperback)



1

2

3



68060: The Fourth Generation in a Heritage

by Roy L. Druian

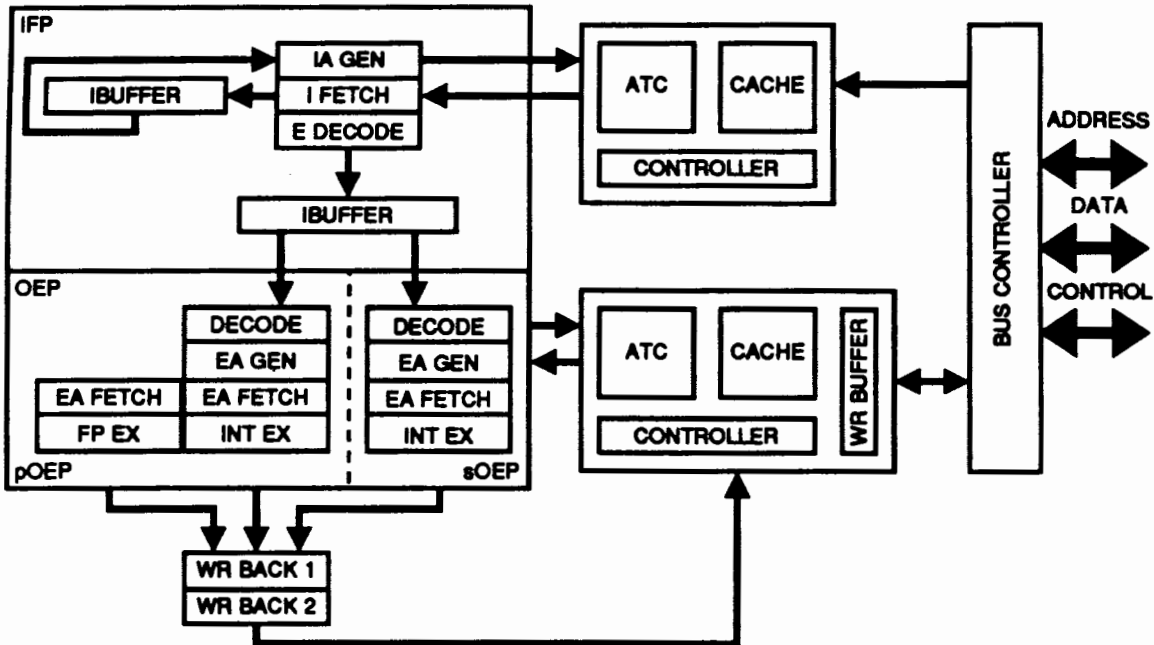
Recently announced at the 1992 Microprocessor Forum is the next high performance 68000 Family member, the 68060. Following a tradition begun with the successful 68000, the 68060 will be the highest performance product that provides full architecture compatibility with all previous members of the product line. The 68060 uses the 68040 feature set as a base model for implementation, with on-board integer, floating point, cache and memory management units. The resemblance however, stops at the programmer's model. Contained within the 68060 is a complete re-implementation of the internal architecture, with the greatest significance being a fully superscalar, superpipelined integer unit, allowing execution of two instructions simultaneously.

Increased performance is key to the development of the 68060. Based on evaluating literally millions of system bus traces, the 68060 provides at least a 1.6 time improvement of the 68040 at the same clock speed using existing compiler technology. The processes technology for the 68060, operating at 3.3V with .5 μ feature sizes, offers initial clock speeds of 50 MHz and 66 MHz allowing the 68060 to deliver 3 - 4 times the performance of the 68040 today. Additionally, compilers can further take advantage of the superscalar operation to provide an additional 15% - 20%.

Below is a block diagram of the 68060. The heart of the processor is the ten stage instruction execution pipeline, where the superscalar operations are performed. The integer unit consists of two main sections, the Instruction Fetch Pipeline (IFP) and the Operand Execution Pipeline (OEP). Multiple instruction issue for the M68000 architecture requires gathering of the multiple words that compose many of the instructions. In order to provide a continuous stream of instructions to the OEP, the IFP consists of four pipeline sections that

- 1) generate next instruction addresses
- 2) fetch the instruction
- 3) perform an early decode to define resources, and
- 4) buffer the instructions to the superscalar execution units.

MC68060 Block Diagram



Feature List

- ☐ Superscalar/Superpipelined
- ☐ Hardware Execution (No Microcode)
- ☐ Full IEEE Floating Point (from 88110)
- ☐ Dual 8Kbyte Caches
- ☐ 256-entry Branch Target Cache
- ☐ Dual 64-entry MMUs
- ☐ 50-66 MHz at First Silicon
- ☐ 3.3V, .05μm, TLM CMOS, Fully Static Design
- ☐ Sample 1Q94. Production 2Q94.

Superpipelining, a concept that uses multiple simple pipeline stages, provides for maximum performance as well as frequency scalability. By keeping each stage simple in function, boundary conditions are much easier to test and there are fewer circuit delays in each stage allowing the migration of speed from the initial frequency of 50 MHz to upwards of 100 MHz as the product matures.

Long pipelines however, can affect performance when a change of program flow occurs due to the latency incurred to refill the pipe stages. To minimize this impact, the 68060 employs a branch cache and prediction mechanism that contains the addresses of 256 of the most recently encountered branches. During the address generation cycle of the pipeline execution, the branch cache is accessed if a branch is to occur. In this case, a taken branch (the most common) will find the destination address in the cache and will be used to fetch the instruction from the instruction cache. This destination instruction is then inserted into the pipeline causing, in effect, a zero clock execution of the branch. Since, in normal code, branches execute quite frequently and tend to be biased toward the taken branch, the branch target cache plays an important role in sustaining high performance.

Instructions that move through the Instruction Fetch Pipeline are eventually placed in an Instruction Buffer that dispatches to the superscalar execution units. The buffer provides a sliding window over the instruction stream by looking for candidate pairs of instructions for execution. A sophisticated dispatch algorithm determines if a pair can be issued to the execution unit based on type of instruction, resources required and instruction dependencies. When these conditions are satisfied, the pair is sent for execution. Should the conditions not be met, only the first instruction is issued for execution and the window then moves over the next candidate pair for evaluation. Studies indicate that 50% - 60% of the instruction stream can be issued in superscalar fashion.

The Operand Execution Pipeline actually consists of two integer (pOEP, sOEP) units and one floating point execution unit. Each of the integer pipelines provides instruction decode, effective address generation, operand fetch and execute functions. Floating point execution is performed as a part of the pOEP but uses the 68040 pre-instruction exception model that allows floating point operations to proceed in parallel with integer operation. Upon completion of the instruction, data is scheduled to return to its destination, either in a register or to memory.

Superscalar operation is supported by a Harvard style cache and memory management subsystem employing separate 8 Kbyte instruction and data caches with 64 entry address translation caches. These large caches work to insure that the most recently used data and instructions are available to the pipeline for immediate execution. With the exception of the larger cache size, this subsystem is modeled directly from the 68040 to allow easy migration of system software to the 68060.

External accesses are made via a bus interface that provides a superset of functionality to the 68040. To accommodate existing 68040 implementations, the 68060 provides a compatibility mode that allows existing ASICs and system designs to be reused. For high speed operation, a superset of functionality, primarily in bus arbitration and cycle termination, is utilized. It is expected that most systems that address the cost effective implementations will utilize the 68040 compatibility options. Packaging for the 68060 will use the standard 68040 Pin Grid Array (PGA) with the currently unused inner row of pins designated for the extra 68060 functionality.

An important consideration for next generation microprocessors is that of power management. It has been noted that this ability, though certainly required for portable and handheld applications, is also important due to the large number of computer systems used in homes and business. The 68060 employs several power management features to aid in minimizing this usage. The design of the 68060 uses a 0.5µ 3.3V process that is fully static. Static operation allows sections of the device that are not currently operating to be shut down, unlike a dynamic implementation that must be continuously clocked to maintain data. Since power is a function of frequency, voltage, and capacitance (number of transistors), minimizing the clocking can be valuable. More important is the reduce voltage requirements as power is actually a function of the square of the voltage. Finally, a new instruction (LPSTOP) allows the 68060 to be placed in a quiescent state when it is not needed to minimize power usage.

The 68060 not only addresses the needs of the next generation computer marketplace but other markets of great significance as well. Due to the wide spread success of both the 68EC0x0 family and the 68300 family that serve the wide variety of embedded and integrated needs, the 68060 design reflects the ability to migrate to these areas as well. The modular design allows removal of the floating point and memory management functions to easily derive 68EC060 and 68LC060 functionality. Additionally, on-chip control bits provide the ability to disable these functions to allow the 68060 to emulate these devices. Furthermore, the design of the 68060 integer core provides a future high performance core in support of the growing 68300 family performance needs.

All in all, the 68060 will provide a growth statement of commitment for the 68000 family bringing competitive levels of performance with the compatibility and ease of use 68000 users have become accustomed to. Scheduled to sample in 4Q93 and be in production during 1994, the 68060 will surely support the needs of the marketplace that exists today, but can serve those applications that will arise from those who today only dream of the next innovation.





Advanced Exec and CPU Issues

by Michael Sinz

As we have seen, the pace of technology changes are getting faster all the time. It is hoped that the Amiga and its applications will be able to keep up with these changes as best as possible. This however means that changes will need to take place in both the Amiga's OS and in application software.

With the release of V37 and V39 Exec, a number of new concepts have been advanced to a stage from which a number of new technologies can be launched. In fact, a number of the functions in V37 Exec are required in order to make the system work with 68040 CPUs in a consistent manner. (Namely the CacheControl(), CachePreDMA(), CachePostDMA(), CacheClearU(), and CacheClearE() calls) With V39, the addition of private memory pools and the cleanup of the memory allocation routines has set the ground work for more advanced memory systems.

New for V39

For V39, we had some time to clean up some of the areas of Exec that could not be fixed in some external manner. The major changes were in the semaphores, memory subsystems, and the ROM debugger.

Semaphores are the key to making a multitasking system work as one system. Exec has a very nice set of semaphore functions that are called SignalSemaphores. Before V39, these semaphores could only be used and accessed in a synchronous manner. That is, you made a function call that would block until you had obtained the semaphore. While this usually ends up being enough for most people, there are times when software may need to "bid" for a semaphore and go and do something until it obtains it. Exec already had this concept in the Procure() and Vacate() functions but these functions were both broken and somewhat useless due to the fact that they worked with a different semaphore structure than the SignalSemaphores. As it turns out, we were able to reuse these two function calls and they now, when used with SignalSemaphores, work and are useful in that the same semaphore may be both synchronously and asynchronously obtained. See the Autodocs on Procure() and Vacate() for more information as to how this works.

One of the features of the Amiga has always been the dynamic nature of its use of memory. This has also been one of the trickiest parts of good Amiga programming. Applications

would like to dynamically use memory and release it but with more than one running at the same time, memory got fragmented and performance suffered. For V39, two different parts were added to the memory subsystem: pools and memory handlers.

Memory pools are a way to help combat memory fragmentation, increase the speed of the system, and give a simple way to keep associated memory together. Due to the fact that memory pools are "private" to the application, a number of performance benefits are obtained (including not needing to go into `Forbid()` during allocation or deallocation from the pool). Since pools give the system a simple way to keep your allocations together it also gives the system a simple way to release your allocations by just deleting the pool as a whole. Also, the design was left blackbox such that future system enhancements can be made to them without too many problems. *(A very important point here is that pools will be the memory interface of choice in the future.)* For more information on memory pools, see the Autodocs and the January/February 1993 AmigaMail article, *Memory Pools*.

Memory handlers are an extension to the Amiga's physical memory management system. As you know, when a memory allocation fails, the system will first attempt to release any resources that are no longer in use and retry the allocation before it will truly fail the allocation. This design was very innovative when the Amiga first came out but it was not complete enough to let applications cache data as long as there was enough memory or to do other, more complex memory usage games.

The memory handler system expands this feature and makes a number of performance problems much easier to deal with. (Such as rasterized outline font caching or large database RAM caching). It also makes it possible for the caching code to know how much and what type of memory the currently failing allocation. This enables the memory handler to intelligently release memory instead of releasing everything. The following is a quick overview of the design goals behind the memory handler design. As a side benefit to this work, most memory allocations are over 100 cycles faster on a 68000 based machine since the overhead of RAMLIB's `SetFunction()` of `AllocMem()` is no longer an issue.

Memory Handler - Quick Overview

The basic design is a handler list that is called when a memory allocation fails. The handler list (just like `input.device`, etc.) contains routines that applications and libraries can access.

Each handler in the list will be called in order until the memory allocation works or the handler list is completed. Only after the handler list has been completely traversed will the allocation fail.

The handler list is a standard Exec-style list that is stored in priority order.

RAMLIB, which currently SetFunctions the AllocMem() routine will no longer do this but rather add itself to the handler list at priority 0. This lets applications come before and after the RAMLIB expunge.

Memory handler related addiitons

There are two new functions in Exec to deal with the handler list and a new flag for AllocMem().

The basic functions are:

```
void AddMemHandler(struct Interrupt *)
                  a1

void RemMemHandler(struct Interrupt *)
                  a1
```

AddMemHandler()

This function takes the handler given and enqueues it onto the memory handler list. Once on the list, the handler must be ready to be called. This means that the handler must be ready to be called before this function even returns.

RemMemHandler()

This function removes the handler from the list. This function *can* be called while within the handler.

A new memory flag, MEMF_NO_EXPUNGE, has been added to Exec. This flag causes the memory allocation attempt to fail without going through the memory handler. This is useful for caching systems that may not really need the memory but will take it if available and also is required for use within the handler such that memory could be allocated during the expunge cycle. (Or at least attempted) This flag will be ignored in systems where there is no memory handler.

```
BITDEF  MEM,NO_EXPUNGE,31 ;AllocMem: Do not call expunge on failure
```

The MemHandler structure

This structure is the data passed to a MemHandler. This structure is *read only*!

```
struct MemHandlerData {
    ULONG memh_RequestSize; /* Size of the requested allocation */           ULONG
    memh_RequestFlags; /* Flags of the requested allocation */              ULONG
    memh_Flags; /* Flags (see below) */
};
```

The memh_RequestSize and memh_RequestFlags are the size and flags arguments from the AllocMem() call that failed.

```
BITDEF MEMH, RECYCLE, 0 ; Recycle
```

The MEMHF_RECYCLE flag is 0 if this was the first time this handler was called due to this allocation failure. If this is 1, the handler is being called again for the same failure. See below about handler return and recycling...

The Handler

The protocol for a MemHandler must be strictly followed. Due to the fact that the handlers are being called on the AllocMem() context and the fact that AllocMem() *must not* break a Forbid(), the handler *must not* break a Forbid(). Another issue is stack usage. The handler could be running on any task in the system that calls AllocMem() For this reason, the handler must try to keep stack usage as low as possible. Exact stack usage is not available, but a good rule would be to keep it under 128 bytes if possible.

The handler may call AllocMem() with the MEMF_NO_EXPUNGE flag. This flag is new to the Exec that has the memory handler system. Library expunge vectors *cannot* make use of this feature. This flag allows a handler to move memory from one location to another. For example, if the requested memory is for Chip, the handler could move any of its Chip allocationst to Fast memory (provided they can be put into Fast memory) and would then be able to help satisfy the MEMF_CHIP request. Also caching systems may wish to only cache an item if memory is available and would not want to have the system do an expunge just to cache this "unimportant" item.

Key points of the handler.

- ☐ The handler will be called in a Forbid() state that *must not* be broken.
- ☐ A handler can RemMemHandler() itself *only* if it returns MEM_DID_NOTHING or MEM_ALL_DONE.

- ❑ The handler code that is in (*is_Code)() of the Interrupt structure will be called as follows:

- a0 = Pointer to (struct MemHandler)
- a1 = Value from is_Data
- a2 = Pointer to the Interrupt structure for this handler
- a6 = ExecBase

The handler must follow the standard rules about register usage. Only d0, d1, a0, and a1 may be modified, all other registers *must* remain unchanged.

Return results:

- d0 = MEM_DID_NOTHING
- or MEM_ALL_DONE
- or MEM_TRY_AGAIN

MEM_DID_NOTHING

If the handler could not release any resources it should return with d0 set to this.

MEM_ALL_DONE

If the handler released all of its resources, it should return this in d0.

MEM_TRY_AGAIN

If the handler released some resources in hopes that it will have solved the memory problem it can return with this value. In that case, Exec will retry the allocation and if it does not work, will call the handler again. Note that the handler can tell if it was already called by the MEMHF_FIRST_TIME flag that will be 0 if this is the first call to the handler. The main use of this return value is to help implement the RAMLIB handler but it could be useful for LRU caching code or caching code that tries to defragment memory during expunge in order to try to satisfy the allocation request.

RAMLIB

RAMLIB will, under this system, no longer SetFunction the memory allocation routines but rather add a memory handler at priority 0. This handler would then be called when the allocation failed and RAMLIB could then call the library expunge vectors as it does today. If RAMLIB wishes to continue to do the 2.0 partial expunge, that would be possible with the MEM_TRY_AGAIN return value.

Simple Amiga Debugging Kernel -- SAD

Another key point in the design of V39 Exec was to provide for a low-level debugging core that can be used to debug rather complex problems. This low-level debugger, the Simple Amiga Debugging kernel, *SAD*, replaces ROM-WACK from pre-V39 systems. One of the goals of SAD was to provide near emulator level access to debugging the Amiga. Due to some minor hardware issues, this was not 100% implemented. The goal was to use the unused NMI interrupt to trap into the SAD kernel and then let the controlling systems talk to SAD and do whatever is needed. By default, due to hardware issues on certain Amiga models, SAD is not connected to the NMI vector. It is ready to be connected, however.

The SAD kernel is a set of very simple control routines stored in the Kickstart ROM that lets debuggers control the Amiga's development environment from the outside. These tools make it possible to do remote machine development/debugging via just the on-board serial port.

Technical Issues

SAD will make use of the motherboard serial port that exists in all Amiga systems. The connection via the serial port allows the system to execute SAD without having any of the system software up and running. (SAD will play with the serial port directly.)

With some minor changes to the Amiga hardware, an NMI-like line could be hooked up to a pin on the serial port. This would allow external control of the machine and would allow the external controller to stop the machine no matter what state it is in. (NMI is that way)

In order to function correctly, SAD requires that some of the Exec CPU control functions work and that ExecBase be valid. Beyond that, SAD does not require the OS to be running.

Command Overview

The basic commands needed to operate SAD are as follows:

- ☐ Read and Write memory as byte, word, and longword.
- ☐ Get the register frame address (contains all registers).
- ☐ JSR to Address Return to system operation (return from interrupt).

These basic routines will let the system do whatever is needed. Since the JSR to address and memory read/write routines can be used to download small sections of code that could be used to do more complex things, this basic command set is thus flexible enough to even replace itself.

Caches will automatically be flushed as needed after each write. (A call to CacheClearU() will be made after the write and before the command done sequence.)

Technical Command Descriptions

Since the communications with SAD is via a serial port, data formats have been defined for minimum overhead while still giving reasonable data reliability. SAD will use the serial port at a default 9600 baud but an external tool can change the serial port's data rate if desired. It would need to make sure that it will be able to reconnect. SAD sets the baud rate to 9600 each time it is entered. However, while within SAD, a simple command to write a WORD to the SERPER register could change the baud rate. This will remain in effect until you exit and re-enter SAD or until you change the register again. (This can be useful if you need to transfer a large amount of data.)

All commands have a basic format that they follow. They all have both an ACK and a completion message.

Basic command format is:

Sender: \$AF <command byte> [<data bytes as needed by command>]

Receiver:

Command ACK: \$00 <command byte>

Command Done: \$1F <command byte> [<data if needed>]

Waiting: \$53 \$41 \$44 \$BF

Waiting when called from Debug(): \$53 \$41 \$44 \$3F

Waiting when in dead-end crash: \$53 \$41 \$44 \$21

The data sequence will be that SAD will emit a \$BF and then wait for a command. If no command is received within two seconds, it will emit \$BF again and loop back. (This is the "heartbeat" of SAD) When called from Debug() and not the NMI hook, SAD will use \$3F as the "heartbeat."

If SAD does not get a response after ten heartbeats, it will return to the system. (Execute an RTS or RTE as needed) This is to prevent a full hang. The debugger at the other end can keep SAD happy by sending a No-Op command.

All I/O in SAD times out. During the transmission of a command, if more than two seconds pass between bytes of data, SAD will time out and return to the prompt. This is mainly to ensure that SAD can never get into an i-loop situation.

Data Structure Issues

While executing in SAD, you may have full access to machine from the CPU standpoint. However, this could also be a problem. It is important to understand that when entered via NMI, many system lists may be in unstable states. (NMI can happen in the middle of the AllocMem() routine or task switch, etc.)

Also, since you are debugging, it is up to you to determine what operations can be done and what cannot be done. A good example is that if you want to write a WORD or LONG, that the address will need to be even on 68000 processors. Also, if you read or write memory that does not exist, you may get a bus error. Following system structures may require that you check the pointers at each step.

When entered via Debug(), you are now running as a task so you will be able to assume some things about system structures. This means that you are not in supervisor state and you can assume that the system is at least not between states. However, remember that since you are debugging the system, bad code could cause data structures to be invalid. Again, standard debugging issues are in play. SAD just gives you the hooks to do whatever you need.

Note: When SAD prompts with \$BF you will be in full disable/forbid state. When \$3F prompting, SAD will only do a Forbid(). It is possible for you to then disable interrupts as needed. This is done such that it is possible to "run" the system from SAD when called with Debug().

Data Frames and the Registers

SAD generates a special data frame that can be used to read what registers contain and to change the contents of the registers. See the entry for GET_CONTEXT_FRAME for more details

For more information on how SAD works, check the Exec Autodocs.

The Future

Now that we have talked about the major changes to Exec for V39, we should look into what the future may bring. What follows is a general description of the "vision" we have for the Exec of the future. This does not mean that exactly these features or all or only these features will be implemented. However, it does show you some of the directions that we hope to be able to push the operating system.

One of the future features that is already somewhat in use is CPU-specific support libraries. Currently, there is a 68040.library that patches itself into Exec and the system to provide the functions needed to make the 68040-based Amiga work. Future processors will also need such support libraries.

A goal will be to have a different library for each of the processor groups. This library would take care of things like handling of the various processor specific issues such as instruction emulation, cache control, MMU support, and other things that are system-level and CPU specific. (In other words, there will be a 68060.library and maybe even a 68030.library)

Along with the CPU-based extensions, the much asked for and very much misunderstood feature of virtual memory will become an issue. The design (from the concept point of view) is rather far along at this point in time and now a number of changes to the memory system will make this possible. In order to prevent compatibility problems and other issues, the only way to obtain virtual memory would be to obtain a private pool with the attribute of PAGED memory. As a side issue, since it is only via private pools that such memory can be allocated, it may be possible to have a form of protected pools, too.

Object oriented programming has become a hot term in the market today. While much of the OOP hype is just that, there are a number of benefits in a system that supports object oriented features. The benefits, however, are only available if there is a good base from which the objects are built and includes the core functions that makes up the basic OOP interfaces.

For 2.0, Jim Mackraz implemented an object oriented gadget/image system for Intuition. Basic Object Oriented Programming System for Intuition or BOOPSI as we call it, was a very important improvement in the dealing with the details of user interface building and operation. The model was designed with those specific goals in mind and it added a great deal to the capability of Intuition and the user interfaces that can be built in Intuition.

One need, however, is for objects that may not be user interface based or have any need for those aspects. Example objects would be a data retrieval object or maybe a network link object or even a "thread" object. In fact, given the correct core set of objects, a full resource

tracking system can be built out of just having objects dispose of their parts when the process or task object is disposed.

The object support that we envision would be very low overhead support for runtime "linked" objects. (Much like shared code libraries are runtime linked.) It would provide for both high-speed method inheritance and complex multiple inheritance. Disk loaded object classes and application embedded private object classes would both be supported.

In addition, the long-awaited task-tree (child-tasking) support is once again on the list of things to do. This would give tasks the ability to be notified about their children tasks (or parent task). Some of this may be well suited to the object-based task or thread construct.

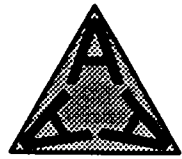
Debugging support will also continue to improve, both via tools such as Enforcer and better support within the new constructs to check for and report invalid operations. The hardest part of developing a major application is the verification of its quality. The system should be able to help here.

Conclusion

So, while much work needs to be done just to keep up with other aspects of the Amiga OS and the hardware (including new CPUs) there are a number of key issues/features that would make for an even better system. Some of these are good because they are great PR (everyone talks about getting VM, even the 1-floppy A1200 owner who does not even have an MMU!) and others are just very useful for system construction and simplified application development.



An Overview of the Advanced Amiga Architecture and Other Future Directions



Document Revision 1.0

1993 Developer's Conference Release

by Dave Haynie
Advanced Amiga System Group

**Copyright ©1993 Commodore International Services Corp, Technology Group
All Rights Reserved**

Information contained herein is the unpublished, confidential and trade secret property of Commodore International Services Corporation, Technology Group. Use, reproduction, or disclosure of this information without the prior explicit written permission of Commodore is strictly prohibited.



Chapter 1

Introduction

Since the introduction of the original Amiga 1000 in 1985, there have been two major revisions of the Amiga chip set: ECS and AA. The ECS chip set introduced a minimally upgraded version of Agnus and Denise, while the AA chip set introduced more significant enhancements, including Lisa, a brand new Denise replacement. Despite the advantages wrought by AA, the AA chip set was still very much an evolution of the original Amiga chip set rather than anything revolutionary. AA maintained most of the original features of Agnus and the entire Paula chip, changing only those things related to video display.

The AAA chip set is the first Amiga chip set to break from this original Amiga architecture. It is composed of four completely new full custom VLSI integrated circuits. It improves every aspect of the Amiga chip set's performance, and its new architecture makes possible many things that could never be directly adapted to the original architecture in any practical sense.

1.1 Targets

This paper is intended to be an overview of the AAA chip set and the direction we're going in with respect to Systems architecture that will surround the AAA chips. This is not intended as a complete definition of either of these, we have hundreds of pages of internal documentation devoted to AAA and next generation systems architecture that does that job.

A good understanding of the original, ECS, and AA chip sets, while probably not vital, will be very helpful in understanding this document and, more importantly, the goals of AAA itself. While there have been many, many improvements in AAA over previous Amiga chip sets, AAA is still very much an Amiga chip set. It is an updated version of many of the same design philosophies that led to prior Amiga implementations.

1.2 Credits

Most of the material on the AAA chip set was gleaned, adapted, and sometimes outright copied from the internal document "Advanced AMIGA Architecture". Many thanks go out to Jim Redfield, Ed Hepler, and the rest of the AAA design group for writing most of this paper for me.



Chapter 2

Goals of the AAA Project

The main point of the AAA chip set is to move the Amiga back toward the leading edge of personal computer technology. However, in doing so, one main requirement is basic compatibility with the existing Amiga chip sets, to keep as much software running as possible without compromising the other AAA goals.

2.1 Compatibility

AAA is designed to be largely register compatible with the ECS chip set. Most of the RGA registers from ECS are supported. The ECS "Ultra hires" registers have been eliminated, as they were never supported in actual practice. Some other display-generation details of ECS are no longer required or supported in AAA.

The AA registers are not supported in AAA. We believe that the 3.0 OS provides the necessary control over AA and that no one need program "to the metal" on AA systems. Additionally, some of the AA features were implemented in a less-than-ideal fashion, in order to fit in the same RGA address space originally implemented in ECS. All AA-equivalent function can be done much better by new AAA support than some kind of AA emulation. Some behaviors can't be perfectly emulated in AAA. Clearly, the AAA chip set's architecture will have an immediate impact on some elements of the ECS emulation apart from register-level compatibility. For example, on a VRAM system, there's no way to slow the system down to an equivalent cycle-stealing ECS mode. So a program will often find significantly more blitter, copper, and CPU access than on an ECS machine. This won't be a problem if you're using the OS correctly, but it could be a problem to take-over-the-system type programs. Such programs may also have some degree of problem with some AAA screen promotions and copper activity.

We envision AAA as a transitional system. It is highly desirable to eliminate the ECS-compatibility, or for that matter, reliance on any register-level compatibility, for the next generation of Amiga chips. So there's an excellent chance that you're seeing some of this stuff for the last time in AAA.

2.2 Flexibility

The pre-ECS, ECS, and to a lesser degree the AA chip sets were each designed with a single system architecture in mind. This defined a good deal of how the chips work together, what kind and how much memory they would support, which CPU interface they'd hook into, etc. Any deviation from this basis could result in a pricey system full of work-around logic, as we had to some extent in the A3000. They also forced this defined Amiga chip sub-system to be the same in all computers, from the low-end to the high-end, so we might have too much expense for an ideal low end, not enough power for an ideal high-end.

The AAA chips are designed to work in several configurations. They can use cheap DRAM, but will gain significant extra performance using VRAM (DRAM with a high speed serial port). They can easily hook up to a variety of 32-bit CPU buses via an easy-to-implement asynchronous slave interface. CPU access to Chip RAM can be gated though similarly to ECS or AA, or the CPU bus can master the entire chip bus (providing all RAM timing) for more efficient CPU to Chip bus access. Finally, the AAA chips can be assembled in "Single" or "Dual" configurations, depending on the display goals.

2.3 Improvements

Every aspect of the previous Amiga chip sets has been improved upon in AAA. While not every feature can be discussed here at length, some of the improvements follow.

2.3.1 Memory Bandwidth

The AAA chip bus is an improvement over the ECS and even AA chip bus in terms of memory speed. The AAA chips run a four cycle burst to Chip RAM, which in raw performance is 4.56 times faster than ECS memory access or 1.14 time faster than AA's two-cycle burst. However, the real key to AAA's memory architecture is its support for VRAM. With VRAM, display fetches have practically no effect on the normal parallel chip RAM bus, freeing it for use by Blitter, Copper, and CPU.

Page-Mode performance is actually a bit better than this implies, since under the right circumstances the AAA chips can run extended burst cycles. Bursts of up to 512 words can be run to keep up with high resolution displays. This improves overall system performance, but increases latency to chip RAM.

2.3.2 CPU Bandwidth

The CPU access to Chip RAM is improved over past systems. Since the AAA chips manage an asynchronous interface to the CPU, CPUs running at any speed take less of a synchronization penalty for chip RAM access than they do in the current A3000 and A4000 systems, where synch-up is managed externally by the Gary chip. Also, AAA's dynamic chip bus slot allocation allows the CPU to get in more often, it's not limited to one out of every two slots. Finally, as mentioned, an external device such as the CPU with some extra support logic can completely master the AAA chip RAM bus, allowing Chip RAM access as fast as today's 32-bit Fast RAM.

2.3.3 Chip Bus DMA

Unlike previous Amiga systems, the AAA chip bus DMA activity is dynamically managed. The different DMA channels (40 of them, including the standard and high-priority external channels for CPU) have fixed priorities with respect to one another, though the relation between the blitter and the CPU can be adjusted in various ways. Because of this dynamic allocation, its possible to run out of cycles before everything has all the time it wants. The

highest priority channels are the deterministic channels, the lowest the blitter or CPU. They are: graphics overlay, DRAM refresh, interrupt transfer, disk, high priority external access, audio, display, sprite, coprocessor, processor, and blitter (the latter two by default and changable, as mentioned).

When too many cycles are requested, something requiring deterministic access has to starve. That something will be the graphics fetch. When the system can't fetch enough graphics data for a line, it sends the graphics overrun interrupt to the CPU. This is to be taken as an indicator to software that something needs to be quieted down.

2.3.4 Blitter

The AAA blitter is significantly faster than the ECS/AA blitter when running in 32-bit mode, thanks to the faster, wider bus. Just doing basic scrolls, it can scroll a 640x200x2 screen about 6 times faster, or a 640x200x4 screen about 9 times faster than with the old blitter. But that's just raw data movement.

Logical improvements to the blitter streamline much of its use. In 32-bit mode, it's much easier to program. It now operates using pixel addressing rather than via masks, modulus, and shifts. It can operate on traditional Amiga bitplanes, or on chunky pixels of 2, 4, 8, or 16 bits width. The line-draw has also been improved, supporting a new "clip-rect" mode for better GUI performance under Intuition. Finally, several arithmetic operations have been added for "sort" and "tally" operations on planes of any pixel depth.

2.3.5 Copper

The copper has been improved in several areas. It can handle 32-bit operations for the new 32-bit registers, and supports a "move-multiple" function for more efficient loading of blocks of consecutive registers, such as color tables. The copper now has an interrupt capability, which lets it receive an interrupt from the blitter. This allows the copper to manage a series of blit operations, one after another, without additional processor intervention.

2.3.6 Graphics

Extensive improvements have been made to the graphics in AAA. A "single" system with Fast-Page DRAM can support displays up to 800x560x9 bitplanes. The same system using VRAM can support 800x560x13 bitplanes or 800x560x24 using "hybrid" pixels. A "dual" system can support up to 1280x1024x5 displays with Fast-Page DRAM, up to 1280x1024x8 bitplanes or 1024x768x24 "hybrid" using VRAM.

The AAA chip set supports 256 CLUT entries of 25 bits each, like AA does. It can handle sprites up to 128 bits wide. It supports up to 16 bitplanes, which makes dual 8-bit playfields possible. The AAA pixel clock is no longer tied to the AAA bus clock, so a variety of display resolutions, even standard ones, can be generated by an AAA system. Hardware-assisted screen promotion is also supported via scaled pixel clocks.

There are a large variety of pixel types supported in AAA. Along with the traditional bitplane-generated pixels (including HAM8 and HAM10), we have several kinds of chunky and compressed pixels.

Half-Chunky	Half-chunky pixels come in 2, 4, or 8-bit depths. These indirect through the color lookup table like most planar modes do.
Chunky	Chunky pixels are 16-bits deep. They bypass the CLUT, providing 5 bits of red, 5-bits of green, 5-bits of blue, and one genlock overlay directly.
Hybrid	Hybrid pixels are 24-bits deep, composed of separate chunky planes for Red, Green, and Blue. The genlock overlay is fixed at {R,G,B} = 0 for this mode.
PACKLUT	These compressed pixels are stored at 2 bits per pixel and decompress to 8-bit half-chunky pixels. This is done by dividing the screen into 4 x 4 pixel regions. Each region contains two colors (8-bit values indexed through the CLUT) and sixteen pixels.
PACKHY	These compressed pixels are stored at 4 bits per pixel and decompress to 24-bit direct pixels. This is done by dividing the screen into 4 x 4 pixel regions. Each region contains two colors (24-bit direct values) and sixteen pixels.

2.3.7 Video Capture

The AAA pixel bus direction can be reversed, allowing an optional low cost video capture device (framegrabber) to be implemented in AAA systems. Capture can be in any chunky display mode. This only works in systems that use VRAM.

2.3.8 Sound

The AAA chip set contains the first improvement in Amiga-based sound since the Amiga was introduced. The audio circuitry can handle sampling rates of better than 50kHz with 16-bit resolution. Eight channels are supported, and channels can be assigned to the left or right output. The 16-bit D/A converters are on-chip, and an external converter is also supported. Additionally, the chip set does 8-bit audio sampling.

2.3.9 Floppy Disk

In addition to supporting the original 1 megabyte disk used in previous Amiga systems, the AAA chip set supports 2 and 4 megabyte disk formats as well. This is, of course, direct support, no speed controls or other kludges are necessary.

As well as supporting the increased floppy densities, the AAA chipset has considerably more flexibility. It has built-in decoding hardware, which can decode MFM, RLL(2,7), and Biphasic Mark (CD-ROM) formats. It can transfer by track, sector, a special "CD mode", and a special high speed track mode. Its data I/O rate has increased from 0.5 Mbit/sec to approximately 11.4 Mbit/sec (though only encoded data can handle this, since the DMA rate available to the floppy logic peaks at 9.9Mbits/sec). Finally, the data separator is programmable, which is very useful at tweaking up to the optimal performance with any specific medium.

This flexible controller can handle practically any floppy format yet invented. With the proper software, it should have no problem with the original Mac format. It handles IBM formats at 360K, 720K, 1.2MB, 1.44MB, and 2.88MB, directly with sectoring (no track buffer necessary). In theory, an RLL(2,7) floppy format at 4 megabyte density could store somewhere between 4.0MB and 5.2MB on a single disk. It should also be able to support 21.6MB flopticals.

And it can support devices other than floppies, too. CD-ROM would be a direct connect, and similarly formatted DAT and digital radio should work too. It might even be possible to support an ST-506 hard drive (assuming such puppies still exist). Higher transfer rates need a cleaner signal, and may require external clocking of the data separator PLL, which is supported by the chip set.

2.3.10 UARTS

The AAA chip set contains two UARTs. Both are improvements over the Paula UART, each buffered with a four-byte FIFO. This significantly improves serial performance at high speeds, since fewer interrupts are taken, and increases reliability, since there's much more time available to respond to a serial interrupt.



Chapter 3

The AAA Chips

The AAA system consists of four completely new VLSI chips, implemented in high speed CMOS. The functions of the first three are partitioned similarly to those of the ECS/AA chip sets. Andrea is the chip bus controller, analogous to Agnus/Alice. Monica is the new display controller, replacing Denise. Mary, the Paula replacement, controls various types of I/O. Finally, a new chip, Linda, double-buffers full display lines.

For the most part, the four chips function as one. Between them, there are nearly 256 word-addressed registers (compatibility registers), 384 longword-addressed registers (new stuff), and 512 longword-addressed CLUT registers. It's often true that a single register address function is performed by two or three of the chips.

3.1 The Andrea Chip

Andrea is the chip bus controller, responsible for managing the chip bus. It is the core of the AAA system, much like a microprocessor is the core of a normal CPU system. The Andrea chip has a rather impressive list of features:

- **Chip RAM control.** Normally, Andrea manages all chip RAM access, supporting both Fast-Page DRAM and VRAM. A simple configuration mechanism tells Andrea what's on the chip bus at startup time. Eight 256Kword banks of chip RAM are supported, and with a little extra logic banks of Fast-Page DRAM and VRAM can be intermixed on a bank-by-bank basis. A "high priority" bus request allows an external device to master the chip RAM bus rather than Andrea.
- **CPU bus gating.** The Andrea chip controls access to the chip RAM bus and chip registers from the CPU port. CPU addresses go through Andrea, CPU data is externally latched under Andrea's control. This is a reasonably general purpose CPU interface, which manages synchronization to the chip bus, directly supporting CPUs of varying clock speeds (previously this was done with extra logic wrapped around the ECS and AA chip sets, like the A3000 and A4000).
- **Chip bus control.** A variety of control signals for management of both the chip bus and various aspects of the other AAA chips are generated in Andrea. The multiplexed chip address/data bus is also mastered by Andrea. This bus contains a register address at the start of a cycle (like the RGA bus on ECS/AA), data in the later stages of a cycle. All of the AAA chip set's DMA channels are addressed by Andrea, which is also responsible for allocating the various DMA channels according to which units are requesting DMA and the priority of each requesting channel.

Clocks. The Andrea chip manages clocking of both the chip bus and the video display. Control lines from Andrea select one of eight possible pixel clock values available at any given time. These can be changed on a line-by-line basis.

Video timing control. All video timing and synchronization signals are created in Andrea. A number of different counters in Andrea generate video synchs, blanking, and the logical controls necessary to support the AAA display. Andrea also manages a light pen input.

Blitter. The Andrea chip contains the Amiga blitter. This blitter can handle the AAA chip RAM bus at full speed and width, but it also has a compatibility mode that lets it handle 16-bit data just like the pre-AAA blitter. In 32-bit bit mode, the blitter has a considerable number of new functions. The new blitter is pixel addressed, it automatically calculates first and last mask, proper shifts, and whether pre-reads are necessary. It supports pixel sizes of 1, 2, 4, 8, or 16 bits, to cover all AAA display modes. It now has "sort" and "tally" functions designed mainly to assist chunky pixel processing. The sort function will run a bubble sort on a plane of pixels, governed by a sort key. The tally function records the number of instances of each byte value in a plane of pixel data. Finally, the new blitter can perform a variety of arithmetic operations on chunky pixels, including addition, averaging, subtraction, saturated subtraction, etc. The following tables illustrate example blitter speed (screens scrolled/second) for various system configuration:

Single, Fast-Page DRAM

Display	640 x 200	640 x 400	800 x 560	1024 x 768	1280 x 1024
2 Bitplane	489.06	233.42	124.54	NA	NA
4 Bitplane	233.37	105.58	51.49	NA	NA
8 Bitplane	105.53	41.65	14.96	NA	NA
16 Bitplane	41.61	NA	NA	NA	NA
Half Chunky	110.22	46.33	19.03	NA	NA
Chunky	46.52	14.59	NA	NA	NA
Hybrid	24.99	NA	NA	NA	NA
ECS 2 Bitplane	81.80	25.83	NA	NA	NA
ECS 4 Bitplane	25.87	NA	NA	NA	NA

Single, Video DRAM

Display	640 x 200	640 x 400	800 x 560	1024 x 768	1280 x 1024
2 Bitplane	504.23	233.42	124.54	NA	NA
4 Bitplane	248.54	105.58	51.49	NA	NA
8 Bitplane	120.70	41.65	14.96	NA	NA
16 Bitplane	56.78	NA	NA	NA	NA
Half Chunky	127.39	46.33	19.03	NA	NA
Chunky	63.70	14.59	NA	NA	NA
Hybrid	42.17	NA	NA	NA	NA
ECS 2 Bitplane	81.80	25.93	NA	NA	NA
ECS 4 Bitplane	25.87	NA	NA	NA	NA

Dual, Fast-Page DRAM

Display	640 x 200	640 x 400	800 x 560	1024 x 768	1280 x 1024
2 Bitplane	498.24	242.59	134.05	71.66	38.56
4 Bitplane	242.56	114.75	61.00	30.05	13.59
8 Bitplane	114.72	50.83	24.48	9.24	NA
16 Bitplane	50.80	18.87	NA	NA	NA
Half Chunky	118.77	54.86	27.72	11.93	3.44
Chunky	55.06	23.12	9.63	3.31	NA
Hybrid	33.54	NA	NA	NA	NA
ECS 2 Bitplane	81.80	25.83	NA	NA	NA
ECS 4 Bitplane	25.87	NA	NA	NA	NA

Dual, Video DRAM

Display	640 x 200	640 x 400	800 x 560	1024 x 768	1280 x 1024
2 Bitplane	504.36	248.69	140.49	78.67	46.38
4 Bitplane	248.68	120.85	67.44	37.06	21.41
8 Bitplane	120.84	56.93	30.91	16.25	8.92
16 Bitplane	56.92	24.97	12.65	5.85	2.68
Half Chunky	126.85	62.93	35.71	20.12	11.92
Chunky	63.36	31.40	17.79	9.99	5.89
Hybrid	41.62	20.32	11.36	6.25	NA
ECS 2 Bitplane	81.80	25.83	NA	NA	NA
ECS 4 Bitplane	25.87	NA	NA	NA	NA

Copper. The AAA copper, as mentioned, supports both 16-bit and 32-bit register operations. A multiple move instruction has been implemented to greatly reduce the overhead of large sequential register movements. A new interrupt mechanism allows the copper to be interrupted. Interrupt sources, in order of priority, are vertical blanking, wait finished, and blitter finished. The blitter finished interrupt allows the copper to re-load the blitter with the next blit operation at the end of the current blitter operation. In this way, the CPU can usually just schedule blit operations in the copper's blitter interrupt routine and get on with other work.

3.2 The Linda Chip

The Linda chip is a display line buffer for the AAA system. Linda provides a great deal of the intelligence behind the AAA display system. While one complete line of display data is being fed to Monica from one of Linda's line buffers, the next line is being fetched into Linda's other line buffer. There are many advantages of this prefetch process:

- Much more efficient Fast-Page display fetch cycles can be run than previously possible. Previous systems use zero or two-cycle Fast-Page fetches, AAA systems can run "burst" cycles hundreds of transfers in length.

- Bitplane alignment can still be on word boundaries (as allowed with ECS), Linda makes any necessary realignments in order to pass longword-aligned data on to Monica. There are, however, various alignment restrictions on some of the new display modes:

Mode	Alignment
Bitplane	16-bit
Half-Chunky	64-bit
Chunky	64-bit
PACKLUT	64-bit
PACKHY	64-bit
Overlay	128-bit
32-bit Sprites	16-bit
64-bit Sprites	64-bit
128-bit Sprites	128-bit

- Rather than Fast-Page memory, Video RAM (VRAM) can be used without requiring the strict alignment requirements typical of most VRAM-based display systems. VRAM allows display fetch overhead to be essentially eliminated from normal chip bus activity.
- The pixel clock is easily decoupled from the bus clock. The chip RAM side of Linda runs at chip RAM bus speed, the Monica side of Linda runs at pixel clock speed.
- Packed pixel formats PACKLUT and PACKHY are decoded here, passing them on to Monica as normal 8-bit half-chunky or 24-bit hybrid pixels, respectively.

3.3 The Monica Chip

Monica is the AAA display controller chip. It takes in display timing data generated by Andrea and graphics data fetched by Linda and from that generates 25-bit digital and analog RGB output (24-bits of color, one of genlock overlay). Monica contains the 256 entry CLUT (color lookup table), HAM mode logic, priority control for playfields/overlay and sprite display, and 8-bit digital to analog converters, one each for Red, Green, and Blue.

HAM (hold and modify) mode is of course the special compact high-color display mode provided in ECS and AA chip sets. Using this mode, Monica can either supply a direct CLUT value or modify a previously displayed value. The five and six-bit modes from ECS, with 4096 color resolution, is supported, as well as the eight-bit mode from ECS and a new ten-bit HAM, which allows a full 24-bits worth of color resolution using only ten bitplanes.

An optional one-bit overlay plane is also supported in Monica for chunky or packed display modes. This requires a VRAM-based chip RAM buffer and the chunky or packed display must be in the odd playfield, while the overlay is fetched based on the even playfield pointer. There are a number of restrictions on the overlay plane when compared to a normal second playfield in the traditional bitplane display.

The actual display capability of any given system is ultimately determined by how fast Monica can be fed pixel data by Linda. Which is, of course, determined by how fast Linda can fetch data from the chip RAM bus. This depends on the type of memory, the size of the system (single or dual), and the display mode (chunky modes tend to allow a more efficient fetch from chip RAM for the same resolution). The following tables show non-interlaced resolutions with different system setups.

Single, Fast-Page DRAM

Display Resolution	Bitplane	Half Chunky	Chunky	Hybrid	Pack LUT	Pack HY
640 x 200	16	yes	yes	yes	yes	yes
704 x 200	16	yes	yes	yes	yes	yes
640 x 400	12	yes	yes	no	yes	yes
800 x 560	10	yes	no	no	yes	no
1024 x 768	NA	NA	NA	NA	NA	NA
1280 x 1024	NA	NA	NA	NA	NA	NA

Single, Video DRAM

Display Resolution	Bitplane	Half Chunky	Chunky	Hybrid	Pack LUT	Pack HY
640 x 200	16	yes	yes	yes	yes	yes
704 x 200	16	yes	yes	yes	yes	yes
640 x 400	16	yes	yes	yes	yes	yes
800 x 560	14	yes	yes	yes	yes	yes
1024 x 768	NA	NA	NA	NA	NA	NA
1280 x 1024	NA	NA	NA	NA	NA	NA

Dual, Fast-Page DRAM

Display Resolution	Bitplane	Half Chunky	Chunky	Hybrid	Pack LUT	Pack HY
640 x 200	16	yes	yes	yes	yes	yes
704 x 200	16	yes	yes	yes	yes	yes
640 x 400	16	yes	yes	yes	yes	yes
800 x 560	13	yes	yes	yes	yes	yes
1024 x 768	8	yes	yes	no	yes	yes
1280 x 1024	5	yes	no	no	yes	no

Dual, Video DRAM

Display Resolution	Bitplane	Half Chunky	Chunky	Hybrid	Pack LUT	Pack HY
640 x 200	16	yes	yes	yes	yes	yes
704 x 200	16	yes	yes	yes	yes	yes
640 x 400	16	yes	yes	yes	yes	yes
800 x 560	16	yes	yes	yes	yes	yes
1024 x 768	11	yes	yes	yes	yes	yes
1280 x 1024	8	yes	yes	no	yes	yes

3.4 The Mary Chip

Mary is the AAA peripheral controller chip. It manages floppy disk, audio, and serial (UART) I/O. It is the first improvement in these areas for Amigas built into an Amiga chip set, and the improvements are considerable.

The floppy disk system was discussed in some detail in Chapter 2. Basically, the data separator now runs up to twenty times faster, with variable parameters and the option of a separate PLL clock. Mary supports internal decode of several formats, several new transfer options, hardware CRC calculation, and a higher DMA bandwidth to chip RAM. A comparison with Paula details these new features:

Function	Paula	Mary
raw bit width	2000, 4000 ns	88-9000 ns
max raw bit rate	0.5 Mbit/sec	11.4 Mbit/sec
encoding methods:		
raw	yes	yes
GCR	yes	yes
MFM	no (done in software)	yes
RLL(2,7)	no	yes
Biphase-Mark	no	yes
sync	16-bit	32, 16, 8-bit
transfer modes:		
track	yes	yes
sector	no	yes
CD digital	no	yes
"trackplus"	no	yes
hardware CRC	no	yes (sector, trackplus)
async PLL clock	no	yes
input types	pulse	pulse, NRZ

Mary's audio system is also a substantial improvement over Paula's. The sampling rate,

number of channels, sample resolution, volume resolution, sampling accuracy, and flexibility have all been improved. On-chip DACs are now at 16-bit resolution, and a digital audio output is available at 20-bits per channel (the supported samples are 16-bits, but sample volume yields a 17-bit result, and eight channels summed into a single output without scaling yields a 20-bit

Function	Paula	Mary
sample rate	29kHz	64kHz
channels	4	8
volume bits	6	12 (signed)
volume aliasing	yes	no
sample size	8 bits	8 or 16 bits
digital out	no	yes
dynamic range	15 bits	16 bits
channels on left	2 (0,3)	0-8 (any or all)
channels on right	2 (1,2)	0-8 (any or all)
global mono bit	no	yes
output time resolution	280 ns	280 ns
period resolution	280 ns	280/64 ns

value). A comparison of audio in Paula versus Mary yields:

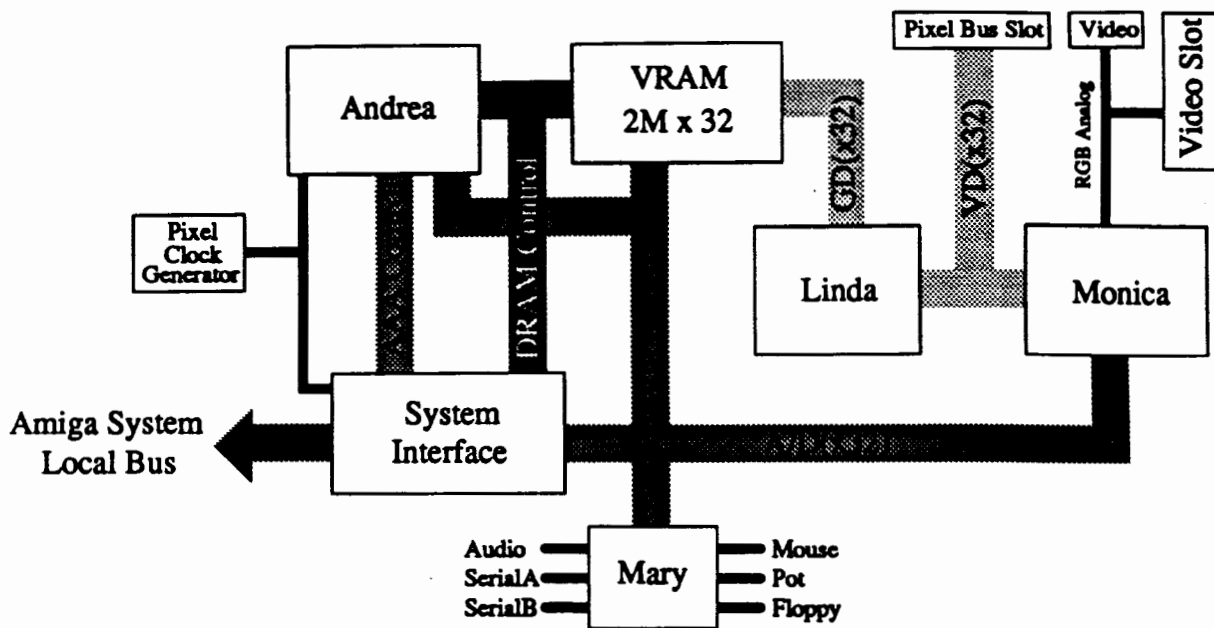
The pot inputs in Mary have been enhanced over Paula's in separate ways. First of all, in order to make them consistent between the widely varying display frequencies supported in AAA, Mary maintains its own sampling counter, which is basically an NTSC compatible horizontal line counter. There is also a new audio sampling mode, which turns the pot lines around to support 8-bit stereo input (with appropriate analog interface circuitry). Higher resolutions are available via an external ADC.

As mentioned before, the Mary UART is an enhancement over the Paula UART. A four deep FIFO has been added, which will reduce interrupt overhead (and overruns caused by missed interrupts) and make faster serial reads possible. Mary contains two UARTS, one that's in the same RGA address as the original Paula UART.

3.5 The Single System

The "single" AAA subsystem consists of one each of the four AAA chips. This example system uses video RAM, which is certainly the preferred implementation. Andrea controls this VRAM and the chip bus. The 32-bit parallel port of the VRAM connects to Andrea and the A/D bus (register address and all data go across this bus). The serial ports of the VRAM connect to Linda, which handles all the video fetching, under Andrea's direction. Mary connects to the A/D bus and to various I/O ports, such as audio, floppy, and serial. Finally, Monica connects to the other side of Linda, the A/D bus, and to the video output (digital or analog).

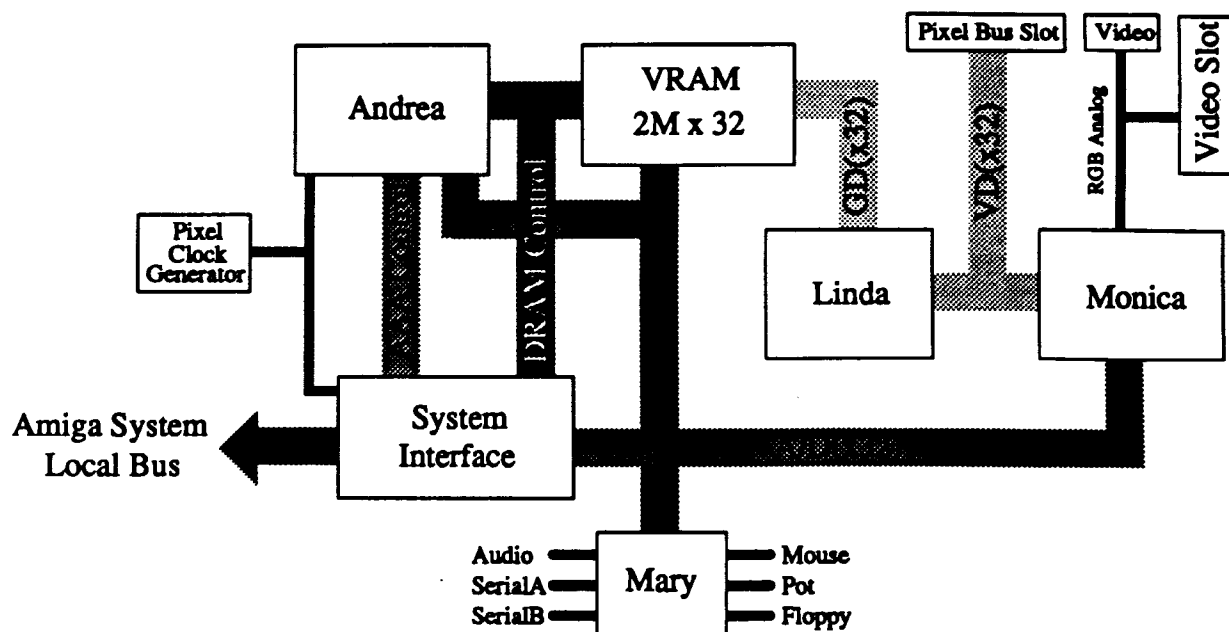
The interface to any kind of system will be implemented in some kind of "glue" chip, most likely a new gate array, though this function certainly could be implemented in a PAL and



TTL based circuit, as on the AAA prototype system. Ideally, such a glue chip will do a variety of jobs. It should buffer data between the CPU/Local bus and the chip bus. A four longword deep buffer here will allow a CPU to write to chip RAM without an immediate delay, and it can also be used to match data rates between the chip and local buses, which are most likely running at different speeds. Andrea requires some kind of external logic to properly drive RAS* lines to each DRAM bank -- this also might be implemented in such a glue device. This will be especially true if it's desirable to build an AAA system that handles VRAM in some kind of SIMM, since a SIMM wouldn't naturally drop into AAA's addressing and AUTOCONFIG scheme. The glue chip may also have something to do with pixel clock generation, though the main pixel clock generator will more than likely be a custom clock synthesis device.

The "pixel bus slot" shown is an expansion slot that makes the pixel bus (video data bus) available for expansion. This isn't the pixel output, but the raw data sent from memory to Monica. The main reason for providing some kind of expansion here is to support the framegrabber mode of AAA. In this mode, Linda reverses direction. Data from the pixel bus (supplied by a video capture bus of some kind, in an appropriate format) goes into Linda and is read out and stored by Andrea. This mode also requires the system to be slaved to an external pixel clock, like in genlock mode.

Single AAA systems can also support a traditional AA-compatible video slot. This can support devices that use either analog video or 25-bit digital video. Not every AA-compatible video device will necessarily work here, of course, since the pixel clock and display rates can be significantly higher than those possible in AA systems. It's not obvious, however, that a new type of video slot would be a better solution, though perhaps some of the reserved pins on the AA slot specification will be used for extra AAA signals in actual AAA implementations.



3.6 The Dual System

An alternate AAA system configuration is the so-called "dual" system. This system uses an Andrea and Mary as before, but contains two Lindas and two Monicas. Andrea, Mary, and Monica still sit on a 32-bit random-access bus, but the display path is now 64-bits wide. Lindas and Monicas are paired in "even" and "odd" groups to process, in turn, even and odd pixels. The pixel rate generated at output is twice the pixel clock rate, so pixel rates as high as 110 MHz can be reached with this kind of system.

In order to support HAM mode, which is of course dependent on past pixels, each Monica indicates its last CLUT access and HAM operation on a special HAM bus, and in turn snoops the HAM bus of the other Monica. In order to generate the full-speed video output and still use the Monica video DACs, each Monica chip has a direction control for its pixel bus. In this setup, the even Monica puts pixel information out, while the odd Monica takes that same information in and feeds it, along with its own pixel data, to the DACs. Both 24-bit values go to the DACs in the same pixel clock.

This system supports a 64-bit pixel bus for video capture and other similar operations. It's possible to support an AA style 25-bit digital video slot as well, but with even more limitations. Since the pixel bus of the odd Monica acts as input, only every other pixel is actually available in digital form on this dual system. In some resolutions, that won't be a problem, in others it will. A full featured digital video port would require an external DAC and either a multiplexer or 48-bit pixel path. A better solution for this kind of thing would be to adopt some kind of digital video transmission protocol as soon as one becomes available.

1

2

3



Chapter 4

Future System Concerns

The AAA chips obviously function as just part of a whole Amiga system. It's certainly possible to build a machine, like the Amiga 500, where the Amiga chips essentially are the whole system. Such a system would be composed of the AAA chips, a microprocessor, memory, some CIAs, analog stuff for audio, and one gate array for "glue". Of course, such a system is rather boring to talk about. It's also somewhat unlikely that early AAA systems will be of this flavor.

Taking the high-end route, there's considerably more to an Amiga system than the custom chips. It would, however, be a mistake to base things directly on past high-end systems like the A3000/A4000. The system architecture of those systems, while fine for the time, is lacking in a number of important areas. The most obvious factor is that the A3000 architecture isn't very modular or flexible. It was designed to support our ideas for the A3000, nothing more. Even in the A4000, extra logic was necessary to push this A3000 architecture in a slightly different direction. On the AAA prototype motherboard, there is extensive high-speed PAL logic necessary to implement a servicable but unsophisticated AAA system.

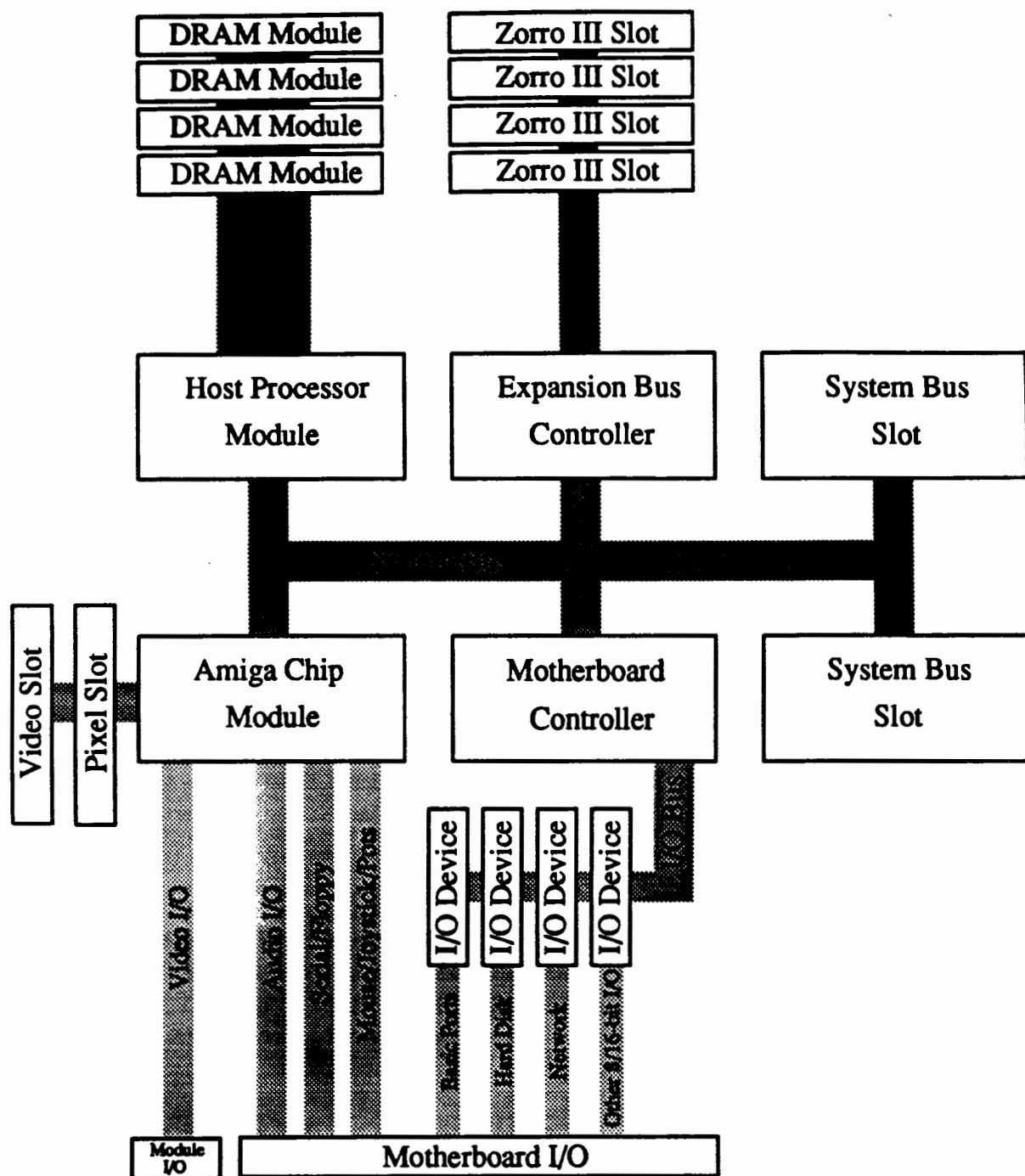
The primary goal of an advanced Amiga system can be summed up in one word: *modularity*. Such a new system, both logically and physically, is composed of several interchangeable subsystems. No one piece has any unnatural dependence on any other; interconnections between the system components have to be based on intentional system standards, not chance implementation details.

4.1 The System Bus

The next generation system should have a processor-independent system bus optimised for chip to chip interconnect. For the most part this replaces the traditional CPU-specific local bus found on previous Amiga systems. This establishes a standard to which several generations of new system and, eventually, Amiga chips can be designed. Since each major system chip hooks into the system bus independently of any other, this finally breaks the interdependence of chips in a chip set, allowing upgrades as necessary to any piece of the system.

4.2 The Motherboard

The motherboard for such a system contains just the basics that will be needed by every system. This will certainly include a number of basic I/O chips for the standard ports on that machine. The CPU, Amiga chips, and various other elements of the system are located on separate modules. These don't necessarily have to be physically located on different cards, but ideally they will be. Not only does this make motherboard upgrade much easier, but it allows several different motherboards to be designed using the same plug-in modules, and it allows Commodore to easily support more options in system and processor makeup.



The heart of the motherboard is the motherboard controller. This manages motherboard based I/O, such as CIA chips, hard disk interface (IDE, SCSI, whatever), network, etc. This also acts as a support for the system bus, handling arbitration of bus master and interrupts. Most I/O chips sit on a flexible 8/16 bit I/O bus defined by the system controller. Many programmable chip selects and interrupt inputs allow new I/O devices to be added as time goes by without extra glue or the need to redesign the motherboard controller.

4.3 The Amiga Chip Module

For the first time in an Amiga system, the Amiga chips are located on a plug-in module rather than fixed into the motherboard. This certainly allows easy upgrade to new Amiga chip sets, but its also a very reasonably thing to consider just in light of the AAA chip set, since a variety of configurations can be made just with AAA, even if no other Amiga chips were considered. The Amiga module connects to the system bus and to a special motherboard-support connector. The system interface device glues Andrea to the system bus, providing the proper bus translation, data FIFOing, and chip selects. The support connector routes general Amiga resources, such as audio, floppy and UART lines, to the appropriate I/O connectors on the motherboard. Digital and analog video also go off this way to mate with video expansion connectors. The main video connector and other connectors specific to the Amiga module will be provided directly on the Amiga module.

4.4 The Host Processor Module

The main system processor is supported on the host processor module. This card has one connector to the system bus, and it gets the first system bus access on boot up by default. It has a second connector to a wide DRAM bus located on the motherboard. While the motherboard houses this DRAM, it's totally up to the host module to drive the DRAM bus. This of course means that the host module must contain a DRAM controller. In most cases, the DRAM controller will be integrated with the system bus controller to provide a low cost interface between host CPU and the motherboard. Locating the DRAM controller on the host module allows DRAM, the most speed-critical element of the system, to be optimized for any host processor chosen.

4.5 The Expansion Controller

The expansion controller is another system bus chip that does conversions to and from the Zorro bus protocols. This can be located on a motherboard (maybe for towers) or on a system bus module (maybe for desktops). It is completely optional -- if a Zorro bus isn't desired, or should be an option on some systems, fine.

4.6 Open System Bus Slots

There will likely be a free system bus slot or two on most expandable machines. These support various kinds of high-speed expansion: fast peripherals, processor farms, DSPs, etc. The number of open system bus slots will of course depend on the final system bus specification, the space available on a given motherboard, etc. Due to the anticipated speed and purpose of such slots, it's not expected that there will be more than two or three open slots in any system, and the module cards aren't likely to be very large. These aren't designed to replace Zorro III as a general purpose expansion bus. Instead, the system bus should be thought of as a more flexible local bus/coprocessor slot.

1

2

3



A1200 HARDWARE DEVELOPER NOTES

*Document Revision 0.9
January 1993*

by George Robbins
Copyright © 1993 Commodore-Amiga, Inc.

Warning: Certain parts of this document are specific to the A1200, while others suggest an expansion connector/architecture that could be implemented in future systems. This document should be viewed only as a statement of design intent, not as a definition of future systems or a commitment that this particular physical/electrical expansion definition will be implemented in any system other than the current A1200.

CHAPTER 1

THE A1200 EXPANSION SLOT

"Revolution is the pod systems rattle from"

-Emily Dickinson

1.1 A1200 System Overview

The A1200 is an Amiga system which includes a 68EC020 processor chip running at 14MHz, the AGA chipset (Alice/Lisa), 2M-bytes of 32-bit Chip memory and 32-bit/120nS system ROM.

By virtue of the 14MHz 68EC020 processor and 32-bit system implementation the system is considerably more powerful than the A500/A600. The inclusion of the AGA chipset both adds new graphics modes and makes more Chip memory cycles available for enhanced performance in traditional display modes.

1.1.2 A1200 Expansion Slot

The overall system arrangement is similar to the A600, but with a full 98-key Amiga keyboard (includes numeric keypad section). The PCMCIA port serves as the primary external expansion interface, and there is also an optional internal 2.5" IDE hard disk. The system board has provision for either 1 or 2M-byte of Chip memory an optional RTC and an optional FPU. There are expansion headers and a hatch in the shield allow installation a small dealer-installable memory/RTC daughter card if the Chip memory or RTC are not configured on the main board.

The major new expansion feature is a CPU bus expansion slot accessible from the bottom of the system which provides both a full processor bus interface and access to various system specific signals. In addition an extra DB25 sized connector position on the rear of the machine has a wiring channel to allow connection of devices in the expansion slot to the outside world.

The new CPU bus expansion slot provides a well-defined, user-installable interface for expansion cards which fast implement Fast memory or faster processors/coprocessors to enhance system performance, or which implement new functions such as DSPs, SCSI or network interfaces or other peripheral devices to extend the basic capabilities of the system.

The pinout of the slot includes signal assignments not only for the 68EC020, but also reserved pins for future 68020/68030 products allowing the potential for expansion cards to serve in several products. This provision should not be taken as an assertion that Commodore will or will not make such products or compatible expansion cards.

1.2 System Implementation Details

The A1200 system is controlled by two gate arrays referred to as Gayle and Budgie. The Gayle array handles all the address decoding and processor control functions, while the Budgie chip implements the 32-bit data paths and DRAM interface.

The Gayle chip is a derivative of the one used in the A600, but has been enhanced to work with a 14MHz processor, 68EC020 control signals and interface to the AGA chipset. The only change to the software model is the redefinition of one of the PCMCIA "speeds" to support zero wait state (16-bit) access for fast SRAM cards.

Unlike the high-end (A3000/A4000) chipsets, the Gayle chip only supports synchronous processor operation at 14MHz and only decodes a 24-bit (16M-byte) address space. While this doesn't preclude use of faster, full-featured processors in the expansion slot, it does impose some additional design constraints to insure that an alternate processor or DMA master in the expansion slot is "well behaved" in Gayle's terms.

1.2.1 Expansion Slot Physical Details

The expansion slot consists of a 150-pin/50 mil card edge connector (male) set into the A1200 system PCB, an internal space provided for the card, and an access door and provision for guiding/retaining the card at the edge connector interface.

The internal space is L-shaped, with a wide area where the mating connector on the expansion board and a narrower tail section that extends out towards the side of the case. The wide section is about 9.1 square inches after allowing for the connector footprint, and the narrower tail section adds another 10.5 square inches.

The design intent of the slot and hatch detail is for a board arranged similar to A601 memory expansion, in that the PCB is upside down, with the major components facing down. However, the vertical space profile above the board is wedge shaped, short in front, with more clearance toward the rear, so that components can be mounted on both sides of the PCB if required.

Overall, the expansion card is not very large and the shape is inconvenient, however given the use of surface mount technology it should be possible to implement some rather sophisticated devices. Trivial devices such as a 4M-byte RAM expansion or 14MHz '030+'882 can fit entirely within the rectangular section, while more ambitious devices may require use of all available area.

The connector space on the rear of the system is slightly larger than a DB25 connector. For use in conjunction with the expansion slot, the connector is mounted on an L-shaped metal bracket which slides into the case from the rear and is secured by one screw coming up from the bottom of the case.

There is an open channel about 2 by 4 inches, to allow routing the cable from the connector to the end of the expansion board. For low pin counts, wires or a small cable can be threaded through, while for higher pin counts, a shredded ribbon-cable or flex-circuit is likely to be required.

1.2.2 Thermal, Power and Electrical Constraints

As with any of the low end systems, there is no explicit provision for cooling - there are vents, but the main cooling mechanism is internal convection and then conduction/radiation from the case surface. As a result there is a practical limitation on how much power can be dissipated in the expansion area without compromising system or expansion device reliability.

This is really not a problem for many devices, however something like a 50MHz '030 or a '040 that requires air flow or a heat-sink is probably unrealistic. In addition, devices intended for sale in the US will require some form of FCC shielding which will only add to the thermal problems.

The A1200 power supply ships with the same de-rated power supply used for the A600. This supply provides 3.0A @ +5V, 500mA @ +12V and 100 mA @ -12V, which provides some margin for a fully loaded system, with all the peripheral ports drawing rated current and drives active. For most users, there is enough current available to meet the expansion power budget specified, if these numbers are exceeded, then the product should be bundled with an A500 replacement power supply to provide adequate margins.

The electrical nature of the expansion connector is basically that of a processor bus access connector, rather than a tightly-specified, well-characterized expansion bus. Expansion devices should put minimal loading and trace length on the signals. Board design should be based on worst case 68EC020 timings, with allowance for genlock deviations.

Critical signals generated on the A1200 usually have series RC termination, tweaked to control overshoot/FCC issues. Critical signals driven by the expansion device should have some provision for adding termination if product verification shows excessive ringing on the A1200 end of the traces.

Due to the constraints of the internal connector position, processor speed and electrical interface, we don't believe that it is reasonable to try to drive an expansion bus or any daisy chained expansion scheme from the expansion connector.

1.3 Fast Processor/Coprocessor Boards

Since the current A1200 is designed around a medium-speed 68EC020, it seems likely that some developers will focus on expansion products that provide a faster processor and/or add a math coprocessor. This should not be too hard to do, however the implementation of Gayle imposes some constraints for proper operation.

Gayle requires that the term: `_AS` asserted and (`_DS` or `R_W` asserted) change synchronously relative to the 14MHz processor clock. A 16MHz 68020 or 68030 running at 14MHz guarantees this. Gayle also requires this term to be false for at least one rising edge of the processor clock between cycles. Failure to meet these criteria will result in internal state machines locking up or cycles being skipped.

In addition, the timing margins for address and data setup/hold are dependent on the setup and hold times of the 14MHz processor timings. If the faster processor does not maintain 14MHz setup and hold times, then the timing specification for the PCMCIA interface, IDE drive and other internal peripheral chips will be violated and end user systems will be unreliable.

There are two basic approaches to implementing a fast processor interface which address these issues. The first is to implement a state machine running at the fast processor clock speed (or 2x) which synchronizes the processor to the 14MHz system clock and issues 14MHz 68020 control signals to the system.

The other, simpler approach is implement a 14MHz state machine which gates the control signals from the fast to the system so they conform to 14MHz timings and also mediates `_DSACK` timing. Either approach will require latching (at least) the data bus, and conditioning of interrupt and/or DMA signals may also be required.

For cycles local to the expansion board, the logic simply insures that Gayle never sees address/data strobe. Since Gayle cycle control and most output signals are conditioned by `_AS` and/or `_DS` the expansion board activity simply appears as idle cycles.

For cycles that access main board resources, `_AS` is clocked over the falling edge of the 14MHz CPU clock, and `_DS` is clocked on the same edge for reads, or the next falling edge for writes. When `_DSACK(*)` from Gayle is clocked in on the falling edge, `_AS` and `_DS` are driven high on the next falling edge, read data is latched and `_DSACK` is passed to the processor. For fast processors, one-shot logic is needed to inhibit `_DSACK` after `_AS` rises or the next cycle may be terminated prematurely.

* `_DSACK` above really refers to `_DSACK0/_DSACK1/_BEER/_HALT` signals.

Since Gayle state machines and read/write strobes run on the positive edge of the 14MHz clock, this protocol insures address/data setup times by virtue of a minimum 35 nS delay on `_AS/_DS` and the 3 state `_DSACK` to address-invalid delay on the fast processor. In addition, clocking this state machine on the falling edge of the 14MHz clock while Gayle clocks on the rising edge effects dual-rank synchronization of the Gayle control signal inputs. Note however, that Gayle's assertion of `_DSACK` is not always synchronous - on zero wait state cycles and on cycles terminated by an asynchronous wait signal (PCMCIA for example), Gayle logic relies on the processor's synchronization of `_DSACK` and the eventual de-assertion of `_AS` to finish the cycle.

Obviously, a fast processor following this protocol can't access A1200 on-board resources any faster than the 14MHz on-board processor. This isn't such a bad thing, since the zero wait state timing (ROM) on the board assumes 14MHz timings. Any real processor speedup has to come from faster processor/coprocessor internal operation, caches or accessing Fast memory or other resources local to the expansion card's fast processor bus.

A well designed fast processor card should take at least all of the 68EC020 control signals into account. Cycle control logic should support bus error and retry cycles. Control signals should be conditioned to 14MHz clock/16MHz processor timings. Be sure to have provision for bus grant to tri-state appropriate control signals to allow compatibility with systems having on-board processor bus DMA masters.

The card must also be somewhat cognizant of the system environment with respect to the bus grant/acknowledge signals. In the A1200 the `_BOSS` signal is simply tied to `_BR` and therefore a processor card asserting `_BOSS` to disable the on-board CPU will always see `_BG` asserted. Future implementations may have functional bus grant mechanisms (including `_BGACK`) to support on-board DMA masters. Assuming the processor card simply grounds `_BOSS`, sensing the state of `_BR` at reset should identify the `_BOSS` implementation.

One trap to watch out for is that the IPL lines which are driven by the interrupt controller in Paula have quite a bit of skew. Even though the 680X0 processor specification claims to treat these as asynchronous inputs, skew can cause a fast processor to split the IPL code, resulting in various spurious interrupt conditions. Reclocking the system IPL outputs on the rising edge of CCK should eliminate this problem, or at least reduce the window to the skew time between the outputs on the flip-flops.

Another issue that must be addressed for processor cards is cache control. The system software assumes that the caches are disabled in some parts of the memory map and then uses the MMU (if available) for finer control. The simple approach is simply to assert cache-in inhibit for accesses to the 16M-byte main board address space.

Alternatively, you can make the ROM and Zorro-II areas cacheable, but some provision to disable cacheing these areas, especially the area shared by the Zorro-II and the PCMCIA main memory would be a good idea.

1.4 Math Coprocessor Boards

While the A1200 as currently configured does not include a math coprocessor, Gayle implements the math coprocessor detect/select protocol and there is a math coprocessor footprint on the circuit board. Since the detect/select signals are duplicated on the expansion connector, adding a 14MHz coprocessor requires simply connecting the appropriate signals.

The implementation of coprocessor cycles in Gayle is to assert the coprocessor select signal, assume that someone out there will eventually assert `_DSACK` and make `_AS` go away. If the detect signal isn't grounded, coprocessor cycles are bus error terminated.

This implementation should support a faster coprocessor running on an asynchronous clock with no additional interface complication. However fast processor/coprocessor combinations should generate their own math coprocessor select signal, since in this case, the timings from Gayle may not meet the `_AS/_CS` constraints as detailed in recent versions of the MC68881/882 manual.

1.5 Memory/Expansion Devices

Since the current A1200 is effectively a Chip memory only system, adding fast, 32-bit memory is the single most effective way of enhancing system performance. While the PCMCIA card does provide some fast memory potential, it is only 16-bits wide and only the relatively expensive fast SRAM cards can support no-wait access.

Because the A1200 implements only a 16 M-byte address space and the software currently implements only the Zorro-II auto-configuration protocol, memory expansion devices are limited to the 4/8 M-bytes allocated to Zorro Space memory expansion in the address map. There are 4-Mbytes available if a PCMCIA card is inserted, 8M-bytes if no card is inserted or the interface is disabled by software.

Memory/expansion device design is similar to that of Zorro bus peripherals, with the exception that the devices should run off the 14MHz processor clock and most memory devices will use the `_OVR` signal to disable the default Gayle cycle termination and assert their own `_DSACK` for 32-bit termination.

All devices should implement the Zorro-II auto-configuration protocol, since this provides for automatic address assignment and device driver linkage. There is a pin on the expansion connector that is defined to be the start of the auto-config chain, currently this is just a ground, but future systems are may put some auto-config resources on the main board and implement this as an output.

There are some fixed decodes implemented for specific devices, (UART, network and RTC) however these should be avoided except for the most trivial devices since system software may assume the presence of specific (but currently undefined) devices at those decodes.

Obviously, the expansion connector has been laid out to support processors with 32-bit address busses and/or faster processor clocks. Since these implementations do not currently exist, it is difficult to completely define their characteristics or the physical/electrical details of their expansion connector.

Two configuration bits are defined on the expansion connector to characterize the system implementation as one of the following:

- ☐ 68EC020, 24-bit address, 14 MHz
- ☐ 68020, 32-bit address, 14 MHz
- ☐ 68(EC)030, 32-bit address, 14 MHz
- ☐ 68(XX)0X0, 32-bit address, > 14 MHz

The intent is really attribute oriented: A 68EC020 system will only drive/interpret 24-bit addresses lines, others support the full 32-bits. A system claiming to be a 68020 will only support `_DSACK` cycle termination, while one claiming to be a 68030 would also support `_STERM` and the cache control/burst signals. 14MHz systems either run the processor at 14MHz or run some expansion cycles with reference to a 14MHz clock. The exception might be a 68030 running the expansion connector at processor speed, but there are other possibilities.

Even though designing for complete upwards compatibility in this sort of environment is problematic, there are a few simple options. A device can operate in either the 24 or 32-bit address environment with logic that checks the 8 high-order address bits, enabled by the configuration code. In the 24-bit environment, these bits are ignored and the device must respond to the Zorro-II auto-config address and protocol, while in the 32-bit environment the device would ignore any cycles where the 8 high-order bits were non-zero.

If future systems/software implement the Zorro-III Autoconfiguration protocol (but not bus protocol) for extended address assignment, then a 24-bit capable card would respond as a Zorro-II card and ignore the Zorro-III auto-config transactions, while a 32-bit capable card could engage in either Zorro-II or Zorro-III auto-config transactions depending on the its requirements and the system type.

Cards purporting to be DMA masters should drive all 32 address lines and the function code lines, even if the 8 high order bits are fixed at zero. There is probably no point in conditioning this based on system type. The A1200 terminates all cycles normally, except for issuing `_BEER` on some PCMCIA access cycles, future systems may implement synchronous termination, retry or bus-timeout options.

The behavior of a card in the exception case (>14MHz) is probably dependent on the card type. If the card is simple and arbitrarily fast, such as a SRAM memory card, a dual-ported SRAM device interface or a simple/fast peripheral chip, it can simply go along with the game.

If it is a DRAM card with internal timing, it might assume the system will honor its cycle termination appropriately, but in this case it would be required to assure that read data is valid no later than one clock after the cycle termination signal, while a DRAM or other card using the processor clock as a critical timing basis would need some sort of jumpers to match behavior to processor clock speed.

The key to providing this degree of upwards compatibility would be fast decode/configuration logic and flexible functional logic, perhaps based on user replaceable PALs or FPGAs. For a simple card like a 4 M-byte memory expansion, anything beyond the 24/32-bit accommodation probably isn't worth the effort and the card should simply refuse to auto-config if it doesn't like the system type code.

CHAPTER 2

PIN AND SIGNAL DESCRIPTIONS

"What can I but enumerate old themes."

-T. S. Eliot

This section contains the pin assignments and signal names for the A1200 Expansion connector. Be sure to read the notes at the end of this section for additional important information.

2.1 68EC020 Processor Signals

A(23:0)	address bus (note 11)
_AS	address strobe
_AVEC	auto vector (note 14)
_BEER	bus error
_BG	bus grant
_BR	bus request (may be tied to _BOSS)
CPUCLK_A	processor clock (14 MHz in A1200)
D(31:0)	data bus
_DS	data strobe
_DSACK_0	data strobe acknowledge
_DSACK_1	" " "
FC(2:0)	function codes (only 1:0 decoded) (note 12)
_HLT	halt request
_IPL(2:0)	interrupt priority (note 14)
_RMC	read/modify cycle (not implemented)
_RST	reset (processor bus)
R_W	read/write
SIZE_0	transfer size
SIZE_1	" "

2.2 Signals not implemented on A1200/68EC020

A(31:24)	processor address lines (note 11)
_BGACK	bus grant acknowledge
_CBACK	cache burst acknowledge
_CBREQ	cache burst request
_CIIN	cache inhibit in
_CIOUT	cache inhibit out
_ECS	early cycle start
_IPEND	interrupt pending
_OCS	operand cycle start
_STERM	synchronous termination

2.3 Math coprocessor signals

_FPU_CS	floating point chip select (note 10)
_FPU_SENSE	floating point chip detect (note 10)

2.4 Amiga system specific signals

_BOSS	main CPU disable (tied to _BG in A1200)
CCK_A	amiga color clock (3.58MHz)
_CC_ENA	credit card enable (note 1)
E	phi-2 clock for 8520's
_CFGOUT	auto-config chain origin
_FLASH	F0-F7 (flash) address decode (use with _OE/_WE) (note 8)
_INT2	priority 2 interrupt request
_INT6	priority 6 interrupt request
_IORD	I/O read strobe (note 2)
_IOWR	I/O write strobe (note 2)
_KB_RESET	power fail reset input
LEFT	left audio channel (note 3)
_NET_CS	D9 (network) chip select (use with _IORD/_IOWR) (note 8)
_OE	memory read strobe (note 2)
_OVR	Gayle decode override (note 4)
_REG	credit card register space (note 2)
_RESET	buffered reset signal
RIGHT	right audio channel (note 3)
_RTC_CS	DC (RTC) chip select (use with _IORD/_IOWR) (note 8)
_SPARE_CS	D8 (UART) chip select (use with _IORD/_IOWR) (note 8)
SYSTEM_0	System type code (note 9)
SYSTEM_1	" "
_WAIT	Wait input for _IORD/_IOWR/_OE/_WE (note 2)
_WE	memory write strobe (note 2)
_WIDE	32-bit termination (broken in A1200) (note 6)

XRDY	wait request (note 5)
_ZORRO	7MHz bus master (not supported in A1200) (note 6)
_xRxD	receive data from serial connector (note 7)
_xTxD	transmit data or'ed to serial connector (note 7)

2.5 Power and Ground

+5V (11 pins)	logic +5 volts @250 MA (???)
GROUND(11 pins)	logic ground
+12V (1 pin)	+12 volts @25 MA (???)
-12V (1 pin)	-12 volts @25 MA (???)
AUDIO (1 pin)	audio ground

2.6 Expansion Connector Notes

1) the credit card **_CC_ENABLE** and **_REG** signals decode to define the two credit card spaces and two address spaces which support Intel/PC strobes and timing.

_CC_ENA	_REG	Address	Description
0	1	60-9F	PCMCIA Main memory (card inserted)
0	0	A0-A1	PCMCIA Attribute (_OE/_WE)
		A2-A3	PCMCIA I/O (_IORD/_IOWR)
1	1	D0-D7	512K PC Memory (_OE/_WE)
1	0	A6-A7	128K PC I/O (_IORD/_IOWR)

2) Various address ranges are decoded and generate Intel/PC style memory read/write (**_OE/_WE**) and I/O read/write (**_IORD/_IOWR**) strobe signals, insert default wait-states for peripheral timing and obey the **_WAIT** signal to add additional wait states.

3) The **LEFT** and **RIGHT** audio and **AUDIO** ground are available on the expansion connector - these are the audio outputs, any output from the card is simply resistively combined with the Amiga audio output.

4) Signals analogous to the Zorro II bus **_OVR**, **XRDY** bus are provided to modify default Gayle operation. Use of these signals is only supported in the Zorro/Auto-config space (20-5F, E8-EF, C0-CF). **XRDY** holds of the assertion of **_DSACK** but does not tri-state the **_DSACK** lines, **_OVR** tri-states the **_DSACK** lines so that the device may assert its own **DSACKS**. **_OVR** must be decoded prior to the assertion of address strobe and is latched for the duration of the cycle.

5) A **_CFGOUT** pin is defined since the A1200 software supports the Zorro-II auto-config protocols. Even though there is no Zorro-II bus, the auto-config protocols provides and effective way of assigning addresses and linking driver software. In the A1200 this pin is grounded, future systems may have on-board auto-config resources which will be configured before asserting this pin.

6) Two other signals are present but not supported - `_ZORRO` modifies `_DSACK` assertion for compatibility with 7MHz bus masters / timing, and `_WIDE` forces 32-bit termination of certain address ranges.

7) The serial transmit and receive data lines from the serial port are present on the connector to allow a high-performance (deep fifo, etc.) UART to take control of the serial port. Using these signals requires a custom serial driver to idle the Amiga UART and talk through the new UART while still using the 8520s to control the RS232 control signals.

8) Various convenience address decodes are present on the connector with nominal function labels. For each decode there is a defined number of wait states and activation of the `_OE/_WE`, `_IORD/_IOWR`, `_WAIT` interface signals. Depending on application requirements you can use either these strobes or the 68020 `_AS/_DS` and `XRDY` signals.

Note that these decodes represent resources in the system address map that may or may not be present in a given system configuration. Any multi-function cards should provide some means of disabling individual functions which might be redundant.

`_FLASH` 512K, 0 wait read/1 write, `_OE/_WE/_WAIT`

Memory at this decode is searched by the software for a diagnostic (early boot takeover) flag/vector and for ROM-tags.

Placing ROM, RAM or FLASH memory at this decode provides potential for application specific ROMs, soft-loading OS modules and other debugging aids.

Since `_WIDE` is broken in the A1200, default termination for this space is 16-bit, asserting `_OVR` disables the decode, so 32-bit memory would have to do its own decode.

`SPARE_CS` 64K, 3 wait read/4 write, `_IORD/_IOWR/_WAIT`

Nominally an Z8530 or INS8250 derivative UART. Note that while slow timing is provided there is no hardware support for the `PCLK` holdoff required by the Z8530 chips, this must be insured by software and/or using one of the derivative chips which minimize this requirement.

`_RTC_CS` 64K, 3 wait read/4 write, `_IORD/_IOWR/_WAIT`

Nominally an OKI MSM6242 or Ricoh RF5C01 Real-Time clock chip. There is provision for this both on board and on an A501-style memory expansion header.

_NET_CS 64K, 0 wait read/1 write, _IORD/_IOWR/_WAIT

Nominally a network interface chip such as the SMC COM2020 Arcnet controller chip or one of the various single chip Ethernet controller chips.

9) A two bit system type code is provided which may be used to identify different main-board environments in case additional systems are implemented which share a compatible internal expansion connector.

The codes do not define planned systems, they are simply best guess at what would make sense in the future. Depending on the level of sophistication, an expansion card might adapt to the environment, provide a diagnostic that it isn't compatible or simply ignore this provision.

System1	System0	Description
0	0	68EC020 (24-bit addr) / 14MHz synchronous
0	1	68020 (32-bit addr) / 14MHz synchronous
1	0	68(EC)030 (32-bit addr) / 14MHz synchronous
1	1	68(XX)0X0 (32-bit addr) / ?MHz asynchronous

What is a safe assumption is that code 00 defines a system which implements a 24-bit address space and any other code 32-bit. This can be used to disable comparison of high-order address bits.

10) The Gayle implementation can meet the timing requirements of a 16MHz MC68881 or MC68882 running off the 14 MHz CPUCLK, however other processor clock/coprocessor clock combinations may run into problems with the coprocessor chip select vs. address strobe timing traps.

The **_FPU_SENSE** line should be treated as bi-directional, there is provision for a FPU on the A1200 board, and this line will be grounded if a FPU is installed. If a coprocessor is present on the expansion card, it should ground this signal.

11) The A1200 only supports 24-bit addressing, that is it decodes and drives A(23:0), while future systems may drive/decode all 32 address lines. From the software point of view, the high order bits are always zero, which is to say that the 24-bit address space maps to the first 16M-bytes of the 32-bit address space.

In a 24-bit environment, external devices must ignore the high order address lines and if a expansion card contains a 32-bit processor with on-board memory outside the 24-bit space, the processor interface must hide accesses outside the 24-bit space from the A1200 system.

12) The A1200 only decodes FC(1:0) to determine the address space, however external processors or DMA masters should drive all three lines. DMA masters should assert FC=0, not FC=3 or leave the lines floating, since FC=3 or FC=7 will be interpreted as a CPU/FPU access.

13) Unimplemented control signals may be un-connected or pulled up to +5V. Implemented control signals will be pulled up to +5V. Unimplemented address lines may be un-connected or terminated. Address and Data buses may unterminated or terminated to Ground, +5 or some other level as required to address FCC/FTZ issues.

14) The _AVEC signal may be actively driven or simply grounded depending on system implementation. As in previous Amiga systems, only auto-vector interrupts are supported.

The IPL lines are outputs from the Paula interrupt controller function - an expansion card should drive either _INT2 or _INT6 to request an interrupt. Fast processors need to reclock the IPL lines with CCK to prevent skew from causing false interrupt level coding.

CHAPTER 3

A1200 EXPANSION FORM FACTORS

"Experience keeps a dear school, but fools will learn in no other"
-Benjamin Franklin

The form factors for an expansion board for the A1200 are shown on the next four pages.

The 150-pin connector used in the A1200 is a 1.27mm (0.050 in.) pitch male edge connector and mates with a female edge connector not previously available:

Fujitsu FCN-225J150-G/A 150-pin version.

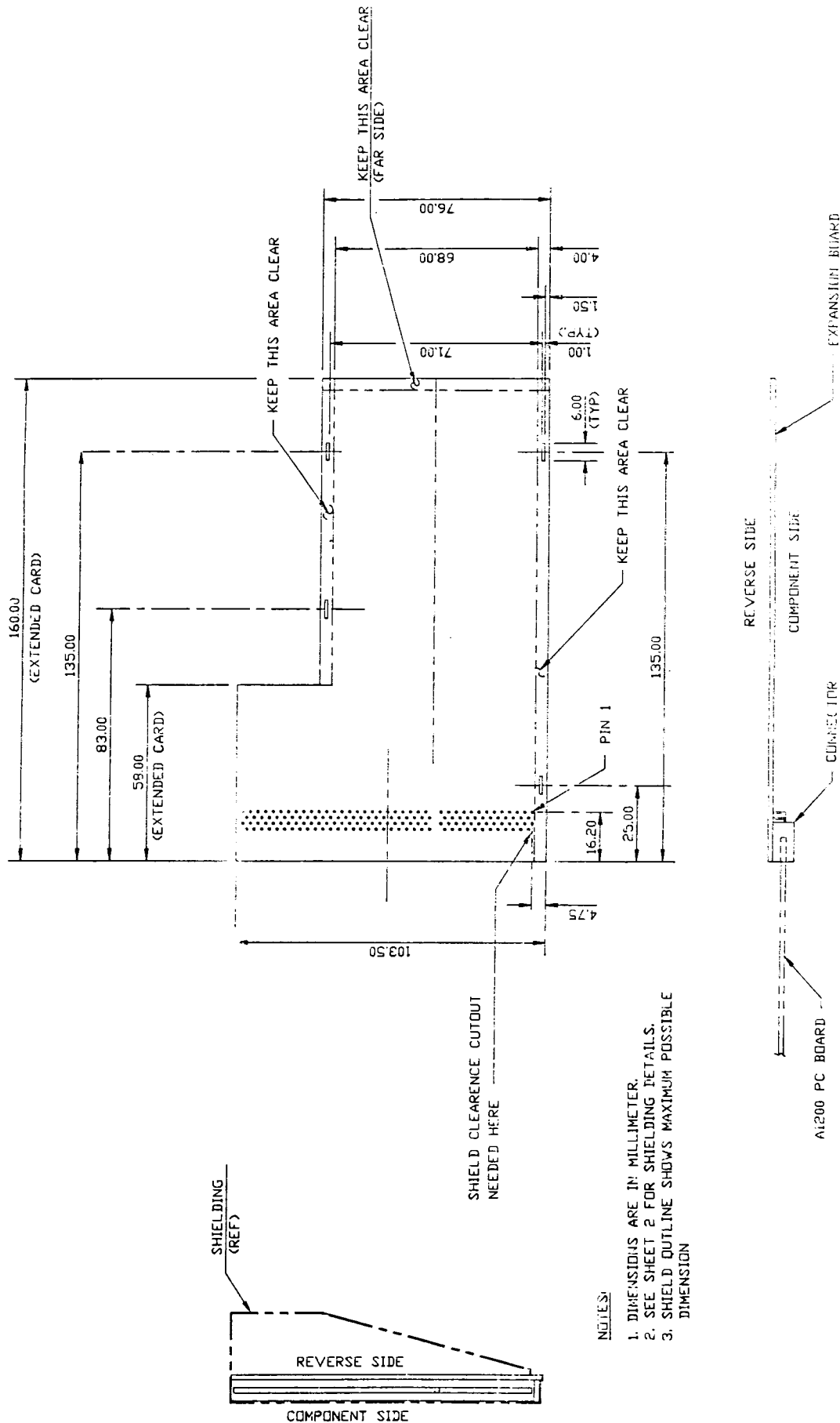
Currently, this part is in short supply. It can be substituted with 2 existing connectors with smaller pin counts. The following 2 connectors can be joined together to provide an interim solution for use in the development of A1200 expansion cards:

Fujitsu FCN-225J050-G/A 50-pin version.

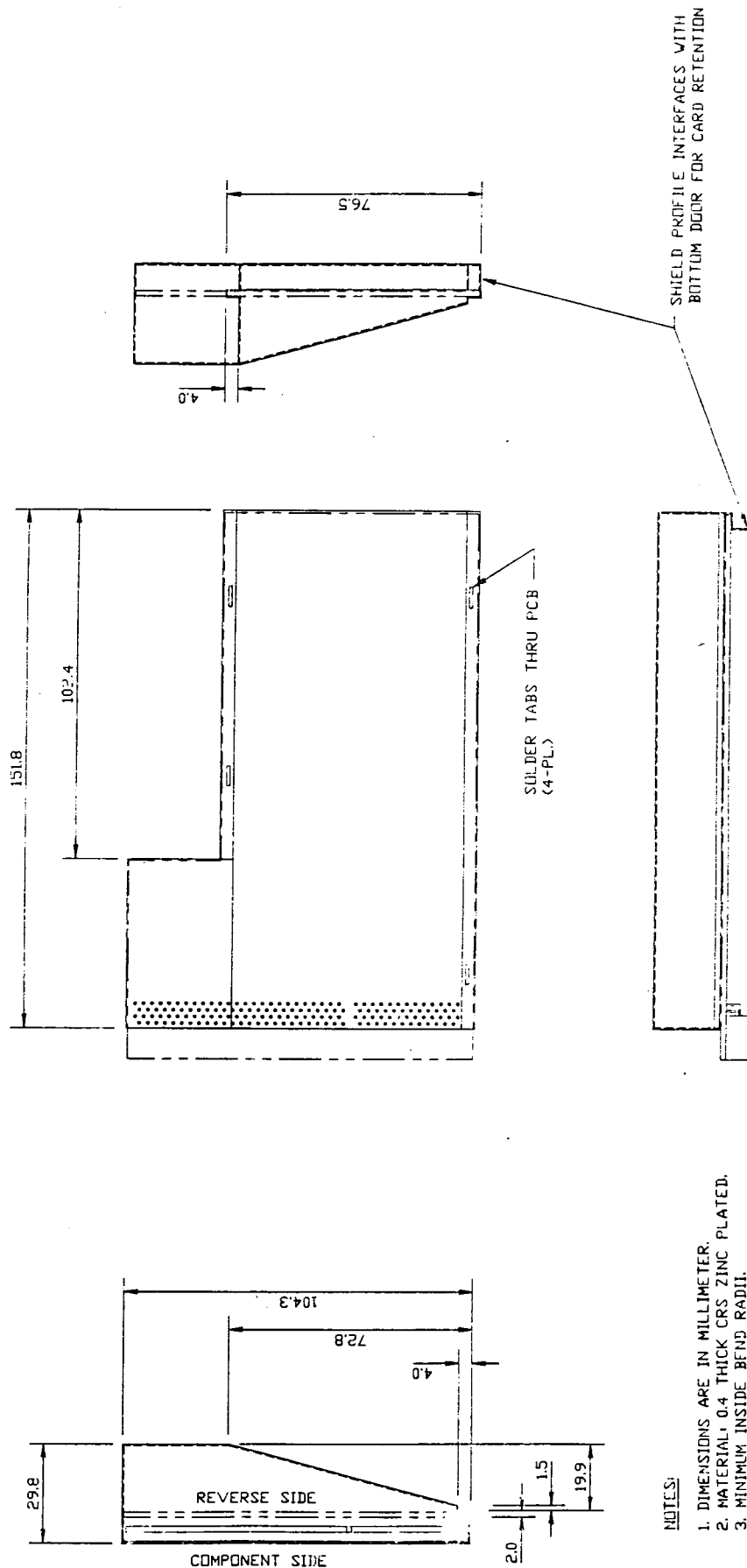
Fujitsu FCN-225J100-G/A 100-pin version

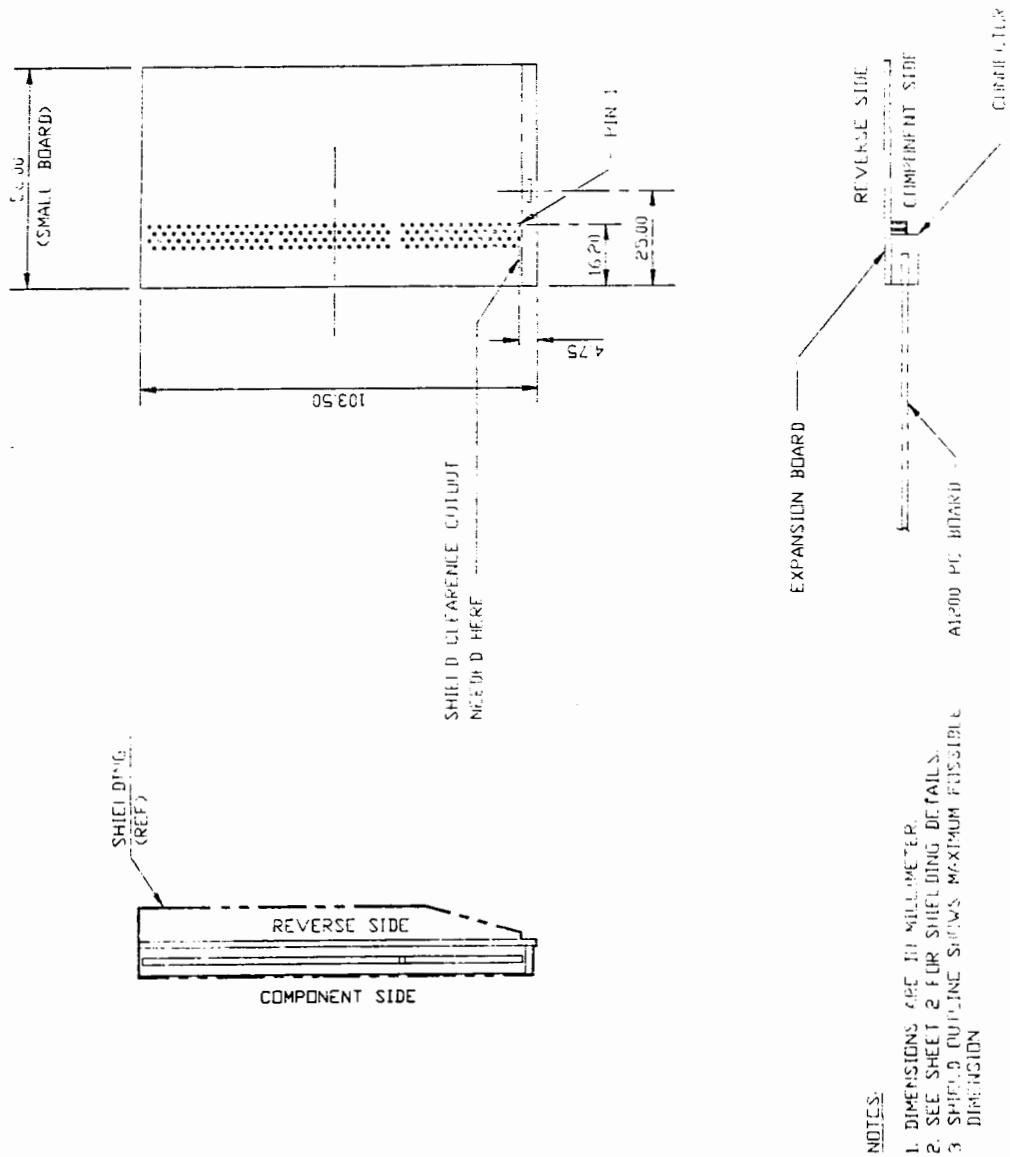
Please note the following if you use the 2-part substitute:

- 1) The pin/row arrangement will not match between a 2-part connector and the real thing.
- 2) Some filing or grinding of the 50 and 100 pin connector housings may be necessary to achieve the same overall length and key/gap distance as the real part.
- 3) Pay strict attention to the orientation of pin 1 and the method for physical mounting of the connector on expansion cards.

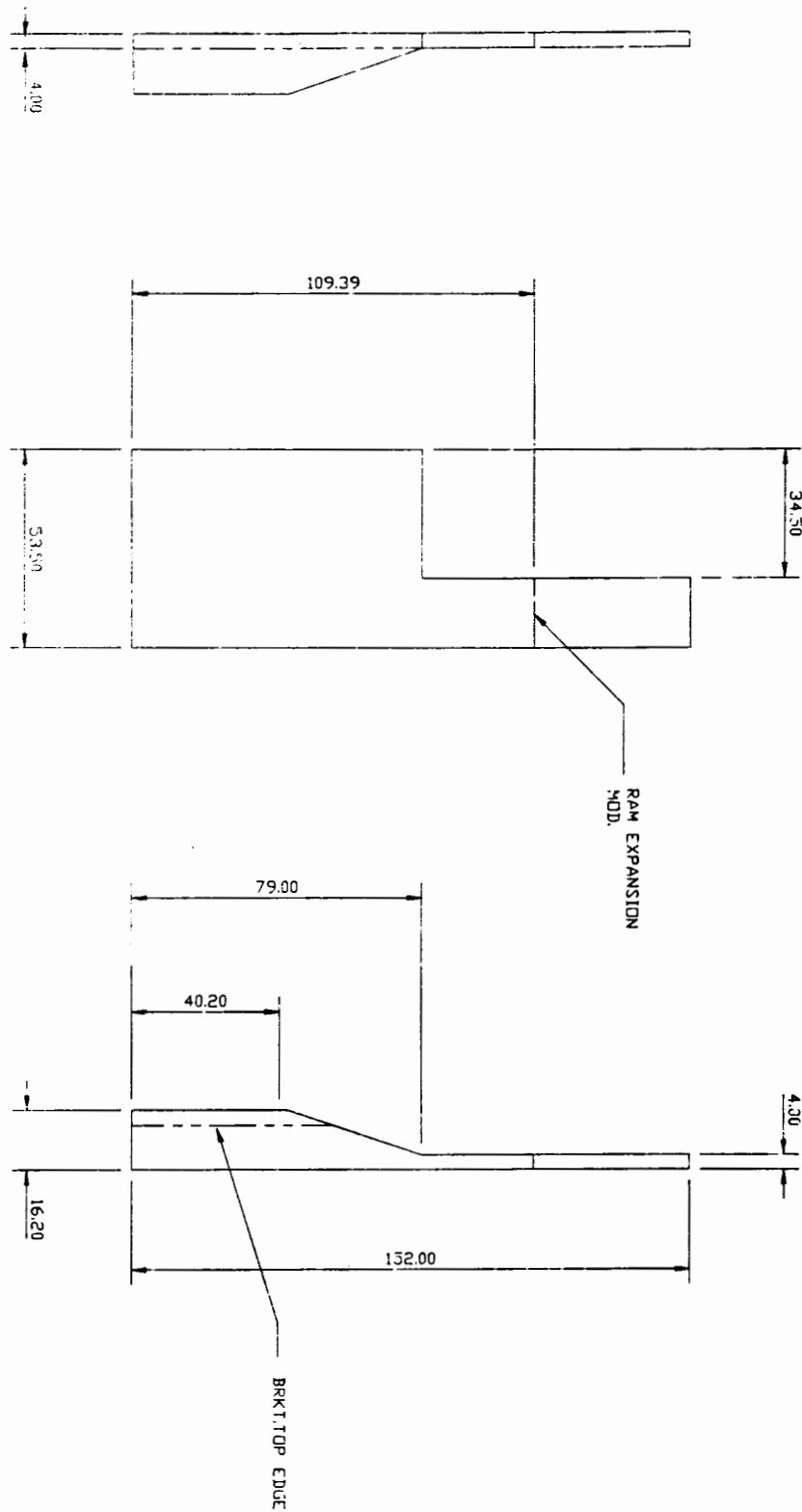


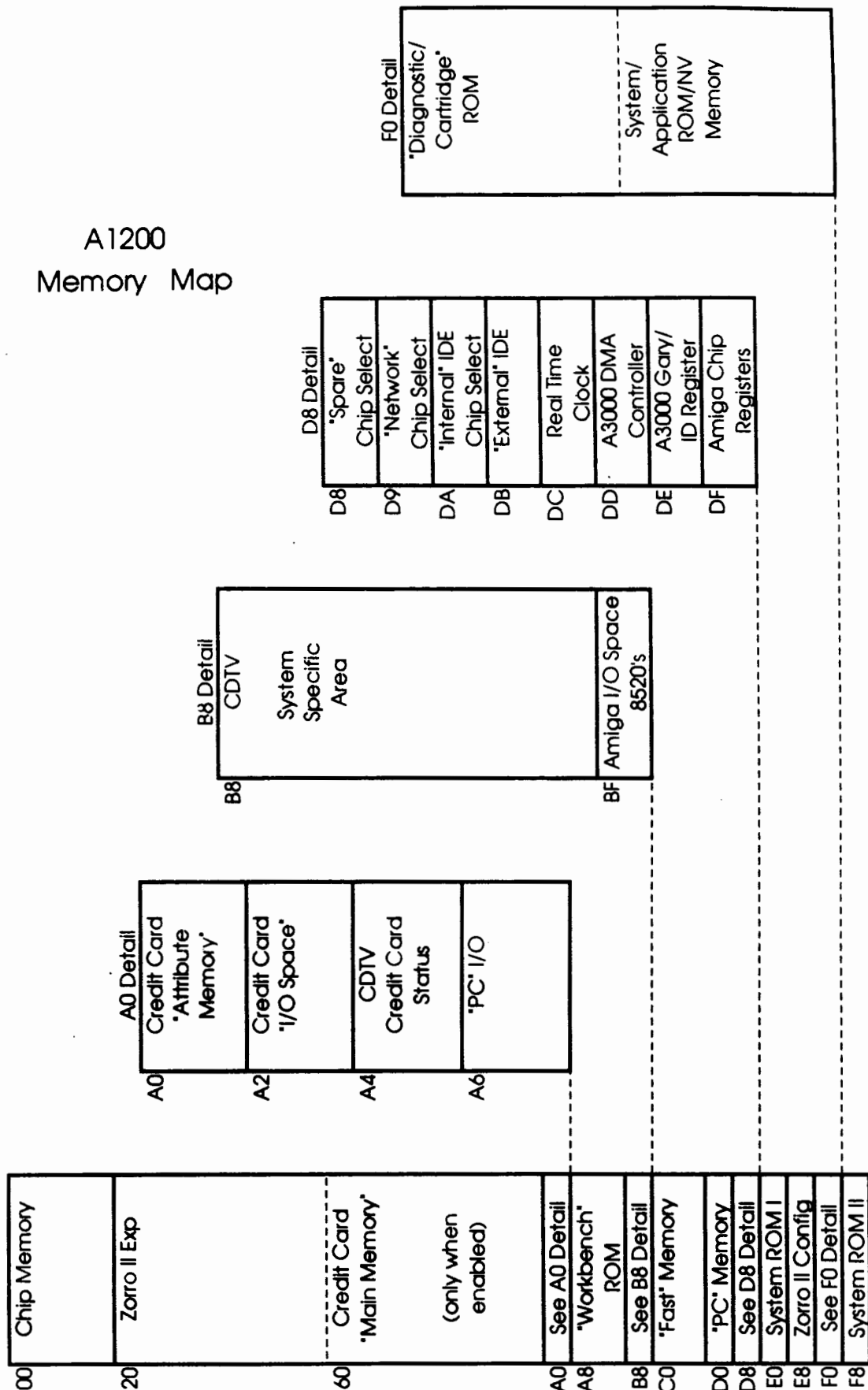
SHIELD OUTLINE SHOWS
MAXIMUM POSSIBLE DIMENSION





Dimensions of Ribbon Cable Channel





A600/A1200 PCMCIA Slot

by Darren M. Greenwald

PCMCIA

PCMCIA stands for "Personal Computer Memory Card International Association." The specification for Release 1.0 was first published in August 1990, followed by the Release 2.0 specification in September 1991. The Amiga 600 PCMCIA implementation was also completed around this time in late 1991. The PCMCIA specification describes card socket physical and electrical requirements, software requirements, and an Intel/MS-DOS specific Execute-In-Place format.

The PCMCIA Release 2.0 specification was followed by the Socket Services Interface Specification Release 1.01, published in September 1991. The Socket Services specification is an Intel/MS-DOS specific proposal that provides low level single or multi socket management.

A PCMCIA socket provides a total of 68 pins including 26 address lines, 16 data lines, two card detect pins, two Vcc pins, two Vpp pins, various status pins, a card RESET pin, a WAIT pin, and other pins required for memory access. A refresh pin is also reserved for use with PSRAM, though not completely defined in the September 1991 Release 2.0 specification. Use of 60-pin and 88-pin Dynamic RAM (DRAM) cards are standardized primarily by EIA/JEDEC and JEIDA, and are not covered by the PCMCIA standard.

PCMCIA cards are available in the Type I format (3.3mm thick), and Type II format (5.0mm thick). In both types, connectors, guides, and other factors are identical. Either type of card may be used in the A600 and A1200 computers.

PCMCIA sockets are designed such that Vcc/Gnd pins connect first when a card is inserted followed by all other pins except for the two card detect pins that connect last (the shortest pins). When a PCMCIA card is inserted, it first receives power, then other pins connect, and finally the two card detect pins connect. Because the two card connect pins are on opposite sides of the card, a stable connection is assured when both of these pins make contact.

The PCMCIA specification defines three memory regions for cards including common memory, attribute memory, and I/O address space. In the Amiga 600/1200 design, access to these memory regions is memory mapped by a custom gate array.

Attribute memory is used to store information describing the card, and may also include configuration registers. Attribute memory is potentially 64 megabytes in size. This is because address generation for attribute memory space is 26 bits wide, plus a REG line. In actual use though most vendors are providing only a few bytes of attribute memory, and partial address decoding to keep costs down. For some RAM cards (and Flash-ROMs) attribute memory is not provided at all.

Common memory is potentially as large as 64 Megabytes. It is common memory space that is used for the contiguous memory provided by SRAM cards, Flash-ROM cards, ROM cards, etc. For cards that do not provide attribute memory, attribute memory accesses are often mapped into common memory space. The PCMCIA software specification allows for this behavior.

I/O address space is memory mapped in the A600/A1200 systems. For both systems, 128K bytes of address space have been reserved for card I/O space. In actual use this has not been a limitation.

PCMCIA connectors have four (4) pins assigned for status information. These are:

- ☐ WRITE-PROTECT (+WP)
- ☐ BATTERY VOLTAGE DETECT 1 (BVD1)
- ☐ BATTERY VOLTAGE DETECT 2 (BVD2)
- ☐ READY/BUSY (+RDY/-BSY)

For I/O cards, the meaning of the BVD1, BVD2, and RDY/BSY pins are interpreted as:

- ☐ BVD2 is used for DIGITAL AUDIO (-SPKR)
- ☐ BVD1 is used for STATUS CHANGE (-STSCHG)
- ☐ RDY/BSY is used for INTERRUPT REQUEST (-IREQ)

PCMCIA digital audio is a simple single-amplitude on-off audio-waveform intended to drive the host's speaker. In the A600/A1200, PCMCIA digital-audio is mixed with Amiga audio output. PCMCIA digital audio is useful for simple sound, such as that provided by modems, but is not suitable for high-fidelity use.

BVD1 and BVD2 are used to monitor battery status for battery backed-up SRAM cards. The combination of these two bits can be interpreted as "battery is good", "battery is low", or "battery has failed." To interpret these pins correctly, the host must be aware that the card is in fact a SRAM card.

PCMCIA METAFORMAT

The PCMCIA Metaformat, or "Card Information Structure" is a software/hardware specification for storing identification information on PCMCIA cards. In theory, every PCMCIA card identifies itself by providing this information in attribute memory. Of course in actual use we find that this applies for I/O cards, but most memory card vendors are not interested in adding to the cost of their cards by providing ROM attribute memory.

The Metaformat is based on a "tuple" structure. A tuple is a simple variable length structure consisting of one (1) identifier byte, one (1) link byte, and 0-255 bytes of data. The tuple link byte tells software how many bytes are contained with the tuple, or at least how many bytes until the next tuple in a chain of tuples. The actual number of valid bytes within a tuple may be less than the link value for the purpose of reserving space.

TUPLE CODE	1
TUPLE LINK	1
TUPLE DATA	0-255 BYTES

The PCMCIA software specification defines numerous reserved tuple codes, an example of which is the required CISTPL_DEVICE tuple. The CISTPL_DEVICE tuple has a code value of one (1). It is required that the first tuple of all PCMCIA cards be a CISTPL_DEVICE tuple. Contained within this tuple we have

DEVICE TYPE	(SRAM, DRAM, I/O, etc.)
DEVICE SPEED	(In Nanoseconds)
DEVICE SIZE	(In units * unit size)

A CISTPL_DEVICE tuple can be as little as three bytes, two of which are required for the tuple code, and link. For example, some I/O cards may only include device type, and device speed interpreted as 100ns, 150ns, 200ns, or 250ns. It is also possible to specify device speed more accurately using one more tuple byte. An additional byte is required to encode device size as 1-32 units of 512 bytes, 2K bytes, 8K bytes, 32K bytes, 128K bytes, 512K bytes, 2M bytes. Using this scheme, it is possible to specify sizes as small as 512 bytes, or as large as 64 megabytes.

You can see that the PCMCIA specification places a strong emphasis on minimizing the amount of data required to store the CIS (Card Information Structure). This makes sense given that adding attribute memory to a card can increase its cost. PCMCIA defines tuples for recording device information, configurable card information, tuple linking, identification of data partitions, manufacturer information, etc.

We will not cover each of these tuples in this talk beyond noting that the definition of each tuple is defined in the PCMCIA Release 2.0 specification. It is worth mentioning that there are many rules that must be observed when searching tuple chains, and a considerable amount of system software can be required to interpret tuple data. To ease the burden, system software in the A600/A1200 provides you with some of the high level functions you will need to read the Card Information Structure of PCMCIA compatible cards.

PCMCIA and the Amiga 600/1200

The Amiga 600 was the first Amiga computer to offer a PCMCIA compatible card slot. The PCMCIA slot on the A600/A1200 complies with Release 1.0 requirements, and supports the Extended Bus Cycle (-WAIT), and Card Reset (+RESET) lines required by Release 2.0. The A600/A1200 PCMCIA slot is based on a memory mapped design, making it possible to support Execute-In-Place code from RAM, ROM, or FLASH-ROM cards.

Internally the A600/A1200 provide PCMCIA slot interfacing via a custom gate array known as GAYLE. GAYLE provides a status register, interrupts for status changes, software control over the Card Reset line, software control over card Vpp, and other features.

Because the A600/A1200 PCMCIA implementation is entirely memory mapped, the maximum common memory address space is limited to 4 megabytes. The upper half of our 8 megabytes of 24-bit expansion space (\$600000 through \$9FFFFFF) has been used for this purpose. Another 128K of address space (\$A00000 through \$A1FFFF) is used for attribute memory, and the next 128K (\$A20000 through \$A3FFFF) are used for I/O address space.

This means that our 24-bit CPU systems can use common memory for Execute-In-Place purposes (SYSTEM RAM expansion, and Execute-In-Place code). The downside is that our current implementation does not yet provide a paging register so that addresses greater than 4MB cannot be generated. While our system software cannot currently support XIP and bank switching well, the extended address space will be useful if we want to support large FLASH-ROM cards. A paging register is being considered for this purpose in the future.

The PCMCIA slot does have some limitations. It is a 16 bit interface, and most SRAM/PSRAM cards are 200-250ns requiring GAYLE provide wait-state generation. A specification for PCMCIA DMA was not finalized for the Release 2.0 specification. We realize that some of you will prefer to use the common memory space for 32 bit RAM expansion in the A1200, so a credit-card disable register is provided in GAYLE. Software modifications have been added to the latest V39 release that automatically disable the PCMCIA interface when RAM is present in card memory space (this does however mean that you lose use of your PCMCIA slot).

Overview of A600/A1200 Software Services

When we designed the A600 PCMCIA software services, our primary concern was to provide a broad range of functionality, and to do so in a friendly way for the end user. This implied we needed to support use of PCMCIA cards as SYSTEM RAM expansion, XIP software distributed on ROM cards, disk-like format cards, auto-booting off disk-like formatted cards, and future support for FLASH-ROM or I/O cards.

The A600/A1200 PCMCIA design was completed before the Socket Services specification was finalized, however we solved many of the issues other vendors are trying to implement now. We felt that the end user should be able to simply plug in a card, and use it without having to configure the card slot for any particular use. It may be that the A600 PCMCIA implementation is the first to offer true "plug-and-play" features. While the design does not lend itself well to a multi-slot support (because of the large amount of contiguous address space required), the user does not have to configure the card slot for a specific purpose, or turn the machine on/off to insert new cards.

The A600/A1200 Kickstart ROM provides low level software services in the form of a resource. We reserve the right to modify the GAYLE hardware, so all GAYLE accesses must be done through this resource. We also provide a trackdisk-like device driver so that disk-like formatted cards can be used.

The A600/A1200 Kickstart ROM provides

☐ **card.resource**

- AutoConfig SRAM/PSRAM cards at boot time as SYSTEM RAM.
- Reset-on-removal of cards in use as SYSTEM RAM.
- Debounce/reset newly inserted cards.
- Hot-insertion/hot-removal/ownership protocol.
- Tuple reader function.
- CISTPL_DEVICE tuple decode function.
- Status change interrupts.
- Card Vpp control (5V or 12V).
- Card reset control.
- Control over hardware reset-on-removal.
- Configure card for I/O interface; enable card digital audio.
- Status register access.
- High-performance, direct CPU access to card memory space.
- Miscellaneous support functions.

❑ carddisk.device

Trackdisk.device like emulation for disk-like cards.
Hot-insertion/hot-removal recognition, like floppies.
Single partition, variable geometry support.
All block sizes, and error detection schemes defined by the PCMCIA 2.0 specification supported.

❑ strap

Autoboot Amiga formatted cards, if installed like floppy disks.
Auto-execution of Amiga formatted Execute-In-Place cards.
Reset-on-removal of active Execute-In-Place cards.
Ability to install device driver software from cards that provide Amiga specific device driver code.

❑ Amiga FileSystem and CrossDOS

Auto-adjust geometry for newly inserted cards.

❑ PrepCard Utility

Prepares SRAM cards for use as SYSTEM RAM.
Prepares/Formats SRAM cards for use as DISK-LIKE media.
Sizes SRAM cards. Auto adjust layout of CIS as required based on attribute memory characteristics.
Displays card information in a human readable way.
Provides advanced settings options.

Overview of the Card.resource Module

Unlike our standard Amiga expansion slots, the PCMCIA slot is unique in that a variety of cards may be inserted/removed when power is on. PCMCIA cards do not have Amiga specific expansion code, so recognition of PCMCIA cards, and configuration must be done outside of expansion.library. The card.resource module provides the low-level configuration protocol, and support functions required for the purpose of configuring PCMCIA cards.

Card.resource functions include

❑ Ownership Functions

OwnCard()
ReleaseCard()

❑ GAYLE/Card Hardware Functions

- ReadCardStatus()
- CardResetCard()
- CardProgramVoltage()
- CardResetRemove()
- CardMiscControl()

❑ Tuple Support Functions

- CopyTuple()
- DeviceTuple()
- IfAmigaXIP()

❑ Flow Support Functions

- BeginCardAccess()
- EndCardAccess()

❑ Miscellaneous Functions

- CardInterface()
- GetCardMap()
- CardAccessSpeed()
- CardForceChange()
- CardChangeCount()

The card.resource module is also responsible for adding SRAM/PSRAM cards as SYSTEM RAM. When a SRAM/PSRAM card is inserted at power-up/reset, card.resource determines if the card is a memory card, and the card size. If the card is not being used to store data, card.resource adds the memory to the system memory list as low-priority fast RAM. When a card is added as SYSTEM RAM, the card.resource module is not added to the system list so PCMCIA device drivers can properly fail to initialize.

Most of the card.resource functions require a pointer to a CardHandle structure defined in <resources/card.h>/<resources/card.i><card.h>.

```
struct CardHandle {  
    struct Node cah_CardNode;  
    struct Interrupt *cah_CardRemoved;  
    struct Interrupt *cah_CardInserted;  
    struct Interrupt *cah_CardStatus;  
    UBYTE cah_CardFlags;  
};
```

❑ **struct Node cah_CardNode**

Used by the resource so that your handle can be added to a list of handles to be notified when a new card is inserted. A complete description of how to fill in this node structure is described in the card.resource autodocs included in your DevCon kit. Note that the priority field of the node structure is used to enqueue handles on the notification list maintained by the card.resource module. Valid priorities for 3rd party device drivers have been pre-assigned. See the autodocs for details.

❑ **struct Interrupt *cah_CardRemoved**

Used by the resource to call an interrupt routine you provide when a card you own is removed. You must call ReleaseCard() to acknowledge that your driver has received the interrupt, and will perform no more accesses to card memory. The card interface is disabled by the resource inhibiting writes/reads of card memory until you acknowledge removal.

❑ **struct Interrupt *cah_CardInserted**

Used by the resource to call an interrupt routine you provide when a card is inserted. Your driver is the owner until you call ReleaseCard(). Usually your driver will signal a task and use the CopyTuple() function to examine the card CIS. If the card is a type your driver understands, you retain ownership of the card until it is removed, or your driver exits. If the card is of a type your driver does not understand, you call ReleaseCard() and the next driver (if any) on the notification list is notified by the resource.

❑ **struct Interrupt *cah_CardStatus**

Used by the resource to call an interrupt routine you provide when a change in the card status register is noticed. Status changes include changes of:

WRITE PROTECT pin
BATTERY VOLTAGE DETECT 1/STATUS CHANGE pin
BATTERY VOLTAGE DETECT 2/DIGITAL AUDIO pin
READY-BUSY/INTERRUPT REQUEST pin

Status change interrupts are optional. If you do not provide a pointer to an interrupt in the handle structure, you do not receive status change interrupts.

□ UBYTE `cah_CardFlags`

Various flags defined in `<resources/card.h>` and `<resources/card.i>`

Card.resource Ownership/Configuration Protocol

`Carddisk.device` is a good example of how an Amiga PCMCIA device driver should be constructed. `Carddisk.device` uses `card.resource` functions to register itself for notification when a new card is inserted. When notified, `carddisk.device` uses the `card.resource CopyTuple()` function to examine the Card Information Structure. If the card contains data stored in disk-like format, `carddisk.device` retains ownership of the card until it is removed.

Other device drivers will be written in a similar way. An example would be a `cardmodem.device` driver that would attempt to obtain card ownership when initialized. Drivers such as this can “hang around”, waiting for an interrupt when a new card is inserted. The CIS contains the information needed to determine if the card is of a type the driver understands, and if not, the driver releases the card for examination by the next driver on the notification list.

Back in September 1991 it became apparent that the PCMCIA specification was still evolving, and that the PCMCIA Metaformat was subject to much interpretation on the part of the card manufacturer. Therefore it did not make sense to try to interpret all tuples, for all cards, and all future cards in system software. This means that `card.resource` leaves interpretation of card CIS up to the individual driver, though a function is provided that handles all the details of walking tuple chains to find/copy specific tuples. Still, even use of this function is not mandatory. Device drivers may provide their own CIS examination code if needed to support a non-compliant card.

The `OwnCard()` function (described in more detail in the autodocs available with your DevCon kits) provides these useful options

`OwnCard()` can be used to “poll” for a card in the slot. This is useful in cases where you need to try for immediate card ownership, but do not wish to hang around if the slot is empty, or in-use. Remember though that when using polling, a card might have just been inserted, and might soon be free for use. If the card slot is not available for use (or empty) your `CardHandle` is *not* enqueued on the notification list.

`OwnCard()` can be used to obtain an interrupt when the card slot is free for use and a card is inserted. This is useful if your driver wishes to hang around and wait without having to poll the resource. Your driver does not

have to check the return value from OwnCard() as you will get an immediate card inserted interrupt if the card is immediately available. If the card slot is in use (or empty) the insert interrupt will come later, if at all.

OwnCard() can return an immediate result of SUCCESS/FAILURE as well as enqueue your handle for subsequent card insertion/removal. This is the default behavior.

OwnCard() can be told to give your driver ownership such that the machine will reset if the card is removed while you own the card. This is not recommended for device drivers (support for hot-removal is preferable), but the option is provided for those that need it.

So there are a couple of ways you can become the owner of the card. First, a successful result from OwnCard() means you are the owner. If a successful result is returned, your CardHandle structure is enqueued on the notification list. The second way is when your driver receives a card inserted interrupt. OwnCard() lets you use the notification method(s) that suit your needs. All GAYLE registers are set to the defaults when you are given ownership of the card slot, or release ownership with the ReleaseCard() function. The defaults are

- ☐ The card interface is enabled.
- ☐ Digital audio is disabled.
- ☐ Hardware write-protect is enabled.
- ☐ Vpp is set to low power 5V.
- ☐ Reset-on-removal is disabled, unless requested by your CardHandle structure.
- ☐ Memory access speed is 250ns.
- ☐ WRITE-PROTECT, BVD1, and RDY-BSY status change interrupts are enabled.

The ReleaseCard() function is used for three purposes

To remove your enqueued CardHandle structure so your driver can exit.

To release ownership of the card when the card is not of a type that your driver understands.

To acknowledge that a card you own was removed.

Normally when you own the card in the card slot your device driver will need to check the Card Information Structure before proceeding. For example, the `carddisk.device` driver checks for a `CISTPL_DEVICE` tuple, a `CISTPL_FORMAT` tuple, and a `CISTPL_GEOMETRY` tuple. If a `CISTPL_FORMAT` tuple is found, `carddisk.device` examines this tuple to verify that the data on the card is stored in disk-like format, determine partition size, block size, error detection method used, etc. The `CISTPL_DEVICE` tuple is examined to determine if the card is writable (SRAM or DRAM), and if not, `carddisk.device` allows reads but not writes. The `CISTPL_GEOMETRY` tuple is examined for the purpose of filling in the disk geometry structure for the `TD_GETGEOMETRY` command.

If you were writing a modem device driver, you would most likely want to check for a `CISTPL_DEVICE` tuple, `CISTPL_VERS_1` tuple (describing the product), and then check for `CISTPL_CFIG` tuples (describing the card configuration registers). During each step your code would be prepared to reject the card as unknown, and release it with the `ReleaseCard()` function.

The `card.resource` module provides you with two useful functions for the purpose of examining the Card Information Structure. The first function is `CopyTuple()`, that scans tuple chains for the tuple you want copied. `CopyTuple()` understands how to follow tuple links, skip odd bytes in attribute memory, and follows the documented exceptions specified in the PCMCIA Release 2.0 documentation. One such exception is that not all cards have a `CISTPL_DEVICE` tuple in attribute memory (in some cases it is not even possible to store this tuple in attribute memory if attribute memory is not provided). `CopyTuple()` deals with this case by looking for a `CISTPL_LINKTARGET` tuple in common memory, and if found, follows the tuple chain in common memory.

`CopyTuple()` also follows implied links, explicit links, skips `CISTPL_NULL` tuples, and has a provision for finding multiple copies of a tuple. Another interesting property of tuples is that the PCMCIA specification allows for storing read sensitive configuration registers in the body of a tuple. `CopyTuple()` was written to avoid reading such registers by allowing you to specify how many bytes of tuple data you want copied.

Once you have a copy of a tuple, you can then safely examine the contents without having to worry about where the tuple is stored in the CIS. Configuration registers, and the offset of other active registers are defined by the CIS.

Another valuable function provided by the `card.resource` module is the `DeviceTuple()` function that takes a copy of a `CISTPL_DEVICE` tuple, and returns device type, device speed, and device size. `DeviceTuple()` understands how to interpret the optional extended device speed byte, how to interpret short `CISTPL_DEVICE` tuples, and will return an error if the `CISTPL_DEVICE` is invalid, or uses undefined bits/byte-combinations.

More on Hot-insertion/Hot-removal

One interesting point to note from the previous sections is that when a card is removed, the card interface is disabled until the card owner acknowledges removal. This means that large data transfer loops do not have to be interleaved with explicit checks for card removal. When the interface is disabled, reads silently return "junk" data, and writes are silently ignored.

The correct procedure is to bracket data transfer loops with the `BeginCardAccess()/EndCardAccess()` functions. If the card is inserted at the beginning of the transfer, but not at the end, your code can conclude that the transfer failed, return an error condition, and call `ReleaseCard()` that reenables the card interface. This procedure also protects against writing to a card inadvertently if a card is removed, and a new card inserted in the middle of your transfer loop (a distinct possibility in a busy multitasking operating system).

Most of the functions provided by the `card.resource` module require you pass in your `CardHandle` structure as an argument. This is used by the resource to verify that your driver is actually the card owner. If you were the owner, but the card has been removed before you have gotten around to calling `ReleaseCard()`, relevant functions above will return errors.

One hot-insertion/hot-removal safety feature of `ReleaseCard()` is that you can safely use this function to remove your `CardHandle` structure from the notification list whether or not you are the card owner. It is possible for a card to be inserted after you call `ReleaseCard()`, but before `ReleaseCard()` actually removes your `CardHandle` structure from the notification list. This should cause you no problem. You may take a card inserted interrupt before your `CardHandle` is removed, but your handle will be removed, and ownership released by the time that `ReleaseCard()` returns to your code.

More Useful Card.resource Functions

Device drivers should use the `GetCardMap()` function to obtain pointers to PCMCIA memory card spaces. Should we want to move these in the future, drivers can continue to work as-is unless we go to a non-memory mapped design.

Device drivers should use the `CardInterface()` function to verify that the GAYLE hardware is present before proceeding. This gives us the option of using a different PCMCIA design for which needed software/hardware workarounds can be published.

`ReadCardStatus()` is used to read the status of the WRITE-PROTECT pin, BVD1/SC pin, BVD2/DA pin, and RDY-BSY/IRQ pin.

CardResetCard() is used to reset configurable cards. The +RESET line is held active for >10 μ s.

CardProgramVoltage() is used to set Vpp to low-power 5V (the default), 5V, or 12V. 12V is required for FLASH-ROM programming.

CardResetRemove() is used to enable/disable hardware reset on card removal. This option is also supported by the **OwnCard()** function.

CardMiscControl() is used to enable DIGITAL AUDIO, and disable GAYLE's hardware write-protect feature (needed for some I/O cards). As of Kickstart V39, 3.01, this function can also be used to selectively enable/disable status change interrupts for changes in BVD1/SC, BVD2/DA, and RDY-BSY/IRQ. More about status change interrupts in the next section.

The **CardChangeCount()** function is used by the system STRAP module to poll for card insertion. This function has been made public in case you require it. The change count is incremented every time a card is removed, and every time a card is successfully inserted.

The **CardForceChange()** function is intended for use by tools, such as PREPCARD to force devices to give up ownership of the card. In general it should not be used by device drivers.

The **CardAccessSpeed()** function is used to set memory access speed (required for wait-state generation). The default is 250ns.

More on Card.resource Status Change Interrupts

GAYLE provides programmable interrupts whenever there is a change in the WRITE-PROTECT pin, BVD2/DA pin, BVD1/SC pin, or RDY-BSY/IRQ pin. For example, **carddisk.device** uses status change interrupts to note changes in the WRITE-PROTECT pin, forcing a disk change so that filesystems automatically send the **TD_PROTSTATUS** command to obtain the write-protect state.

Up through **card.resource V37**, the resource enabled status change interrupts for changes in WRITE-PROTECT, BVD1/SC, and RDY-BSY/IRQ. As of V39 3.01, the **CardMiscControl()** function can be used to selectively enable/disable status change interrupts for BVD2/DA, BVD1/SC, and RDY-BSY/IRQ. WRITE-PROTECT status change interrupts are always enabled, and rare.

You will note that changes in BVD2/DA are not enabled by default. This is because

monitoring for changes in BVD2/BVD1 via interrupts is not that useful since such a tool must also know that card in the slot is an SRAM card. Generating interrupts for I/O cards that provide DIGITAL AUDIO is probably pointless, however should you find some use for monitoring DA changes via interrupts Kickstart V39 3.01 now provides this feature.

The more observant will note that GAYLE generates an interrupt on status change, not just when a status line goes from the inactive to the active state. This is different from normal Amiga hardware that generates level-mode interrupts, and latches the interrupt until the hardware is serviced.

It turns out that even though PCMCIA specification recommends that card vendors support both level-mode, and pulsed-mode interrupts, PCMCIA products will most likely be designed for and tested on Intel based machines (that by default use positive edge sensitive interrupts). To work around this, GAYLE provides positive, and negative edge sensitive interrupts, but the IRQ line from the card slot is not directly connected to PAULA. Amiga PCMCIA device drivers must be prepared for interrupts generated by GAYLE whenever IRQ goes active, or inactive. This means that device driver code may need to test a status register on the PCMCIA card, or if necessary, check the state of the IRQ line using the ReadCardStatus() function.

The card.resource module clears the interrupt on GAYLE when you return from the status change interrupt, however in retrospect we realized that for some cards it is preferable to service GAYLE first followed by the PCMCIA card hardware. This capability is now supported in version 39.1 of the card.resource module, and can be worked around in previous versions of the resource by using exec/AddIntServer() to install an INTB_PORTS interrupt server of priority 127 (causing your interrupt server to be enqueued after the resource's interrupt server). Within your status change interrupt you would set a flag(s) indicating what status bits have changed, and then service the PCMCIA hardware in your interrupt server.

Special Note

Interrupt servers of priority 100 or greater must *not* terminate the server chain. Despite previous publications, some servers require the ability to cause interrupts by poking PAULA. Such interrupts are not latched, and can be missed if the server chain is terminated by higher priority servers. Therefore we now reserve interrupt server priorities 100 or greater for servers that never terminate the server chain.

About Carddisk.device

Carddisk.device emulates trackdisk.device, though a few trackdisk.device commands are not supported. These include

TD_GETDRIVETYPE for which defined types are meaningless for PCMCIA cards.

TD_GETNUMTRACKS is meaningless for PCMCIA cards; use TD_GETGEOMETRY.

TD_RAWREAD and TD_RAWWRITE are not supported; use CMD_READ and CMD_WRITE.

Carddisk.device supports all block sizes specified by the PCMCIA specification, though 128 bytes per sector is not supported by the Amiga FileSystem. Carddisk.device supports disk-like cards that do not use error detection (the recommended PCMCIA default), single-byte checksum error detection, and double-byte CRC error detection.

So far we have not seen MS-DOS PCMCIA implementations make use of block sizes other than the default 512 bytes per block, or use error detection. For maximum data portability we suggest using the defaults when preparing cards with the PREPCARD utility.

Carddisk.device does not support the MS-DOS specific pseudo-floppy format, that does not use PCMCIA tuples to identify disk-like cards. This could be supported by writing a disk-loadable driver that has code to interpret geometry parameters in the MS-DOS boot sector.

Carddisk.device mounts itself as device CC0:, priority 3 which is less than the priority of DF0:, but typically higher than other media. To save memory, the filesystem for CC0: is not started until a disk-like card is inserted in the machine.

Amiga Execute-in-place (XIP) Cards

The A600/A1200 support a simple XIP scheme, however remember that XIP software cannot be removed without causing a machine reset. Because of this, XIP makes the most sense for games, or custom applications. Complete details of how to implement an XIP card are available in the card.resource autodocs.

XIP software is polled for by the system STRAP module at boot time. On systems that do

not have harddrives, the user may insert XIP software instead of a floppy disk when prompted by the STRAP screen. XIP cards take priority over disk-like media, so XIP cards can be booted on systems with harddrives, or removable media.

One thing you will want to note about XIP software is that XIP code may not initialize DOS, or use DOS functions. This is because DOS requires disk-like media to boot from, and in the case of boot-block games it is often preferred that DOS not be started. We do realize that some of you will want the ability to use DOS functions, so a work-around (kludge) is described in the card.resource autodocs.

Back when we designed the A600 we decided that it is unrealistic to expect game developers to write their code to be entirely PC relative. Of course this means we cannot move the PCMCIA common memory address in future low-end machines, but it does allow game developers to use standard compilation techniques, and LoadSeg() game software into SRAM cards for testing. CATS has such a tool, *loadc*, and is making it available in your DevCon kits.

A final point worth mentioning is that the XIP implementation makes it possible for Amiga specific PCMCIA cards to download software into RAM, or install software that executes out of PCMCIA ROM. XIP software can return control to the STRAP module, giving us the option of booting off harddrive or floppy-disk after card specific code has been initialized. This makes it possible to use the PCMCIA slot for a variety of custom purposes.

PCMCIA Attribute Memory

Tuples that appear in attribute memory must use even byte addresses only. This was done to decrease costs, but it turns out that real way to decrease cost is not to provide attribute memory at all. In fact this is what many SRAM, and FLASH-ROM vendors have decided to do. Many cards ignore the REG line, so attribute memory transparently accesses common memory. The PCMCIA specification allows for this behavior, describing the preferred way to arrange the CIS for cards that do not have attribute memory.

Other SRAM cards provide a few bytes of battery backed up attribute memory, and some decode attribute memory space but return junk data (usually \$FF) if the attribute memory EPROM option is not installed. We have yet to actually come across an SRAM card, or FLASH-ROM card that provides ROM attribute memory. What this means is that tools are required (like the PREPCARD tool) that know how to size memory cards, and lay out the CIS depending on the type of attribute memory characteristics of the card.

The more observant may have noticed that the PCMCIA specification requires the first tuple

of all compliant cards be the CISTPL_DEVICE tuple describing device type, device speed, and device size. This is not possible on cards that decode attribute memory space, but return junk data. Therefore our implementation allows for storing the CISTPL_DEVICE tuple in common memory if it is not possible to write to attribute memory on the card. This workaround is not mentioned in the PCMCIA Release 2.0 specification, but seems to be a reasonable one.

Another significant problem not addressed by the PCMCIA specification is mastering requirements for ROM card vendors. If you have an interest in distributing software on masked ROM cards, anticipate that you will need to work closely with the vendor to define your needs. You will need to make sure that the ROM vendor understands that the Card Information Structure must be accurately duplicated, and possibly re-laid out if the ROM vendor's cards have different attribute memory characteristics than what you provide as a master.

PCMCIA maintains a list of product vendors that can be obtained along with the PCMCIA specification. A copy of the PCMCIA specification, and vendor listing is available through CATS (you must contact CATS to obtain a current price for the documents).



1

2

3



A4000/A3000 HARDWARE DEVELOPER NOTES

PART I: THE ZORRO III BUS SPECIFICATION

PART II: THE AMIGA LOCAL BUS SLOT

PART III: THE AMIGA VIDEO SLOT

*Document Revision 1.11
December 1992*

**by Dave Haynie
with Scott Schaeffer, Scott Hood and Dan Baker**

Copyright © 1990, 1991, 1992 Commodore-Amiga, Inc.

1

2

3

IMPORTANT INFORMATION

"A life spent making mistakes is not only more honorable but more useful than a life spent doing nothing."

-George Bernard Shaw

This Document Contains Preliminary Information

The information contained here is preliminary in nature and subject to possible errors and omissions. Few Zorro III cards have yet been designed, so some features described here have not actually been tested in a system, or in some cases, actually implemented as of this writing. That, of course, is one major reason for having a specification in the first place.

Commodore Technology reserves the right to correct any mistake, error, omission (or vicious lie). Corrections will be published as updates to this document, which will be released as necessary. Revisions will be tracked via the revision number that appears on the front cover. New revisions will always list the corrections up front, and developers will be kept up to date on released revisions via the normal CATS channels.

All information herein is Copyright © 1990, 1991, 1992 by Commodore-Amiga, Inc., and may not be reproduced in any form without permission.

1

2

3

ACKNOWLEDGEMENTS

"Art is I; science is we."

-Claude Bernard

I'd like to acknowledge the following people and groups, without whom this new stuff would have been impossible:

- The original Amiga designers, for designing the first microcomputer bus with support for multiple masters, software board configuration, and room to grow.
- The rest of the A3000 Engineers: Greg Berlin, Hedley Davis, Scott Hood, and Scott Schaeffer; PCB master Terry Fisher; and the lab maniacs George Terbush, Brian Fenimore, and Dan Faust. And of course the A3000 boss men, Jeff Porter and Henri Ruben, who let it all happen.
- The folks who helped review the original version of this document, overnight: Joe Augenbraun, Dan Baker, Hedley Davis, Bryce Nesbitt, and Jeff Porter. And to the numerous folks who've helped out with questions, corrections, and other feedback ever since.
- The Commodore-Amiga software group, and the Commodore Semiconductor Group, for excellent support in their respective areas.
- Commodore's Developer Support people from both sides of the Atlantic.
- Gold Disk, for some good and relatively bug free electronic publishing software.
- Iggy; an excellent cat, an excellent foot warmer.

1

2

3

TABLE OF CONTENTS

PART I: THE ZORRO III BUS SPECIFICATION

CHAPTER 1 INTRODUCTION

1.1	Intended Audience.....	1
1.2	Bug Reports.....	2
1.3	Amiga Bus History.....	2
1.4	The Zorro III Rationale.....	3
1.5	Document Revision History.....	4
1.5.1	Changes for Rev 0.90.....	5
1.5.2	Changes for Rev 0.91.....	5
1.5.3	Changes for Rev 1.00.....	5
1.5.4	Changes for Rev 1.01.....	5
1.5.5	Changes for Rev 1.10.....	5
1.5.6	Changes for Rev 1.11.....	6

CHAPTER 2 ZORRO II COMPATIBILITY

2.1	Changes From The A2000 Bus.....	7
2.1.1	6800 Bus Interface.....	8
2.1.2	Bus Memory Mapping and Cache Support.....	8
2.1.3	Bus Synchronization Delays.....	9
2.1.4	Zorro II Master Access to Local Slaves.....	9
2.1.5	Bus Arbitration and Fairness.....	9
2.1.6	Intelligent Cycle Spacing.....	9
2.1.7	Bus Drive and Termination.....	10
2.1.8	DMA Latency and Overlap.....	10
2.1.9	Power Supply Differences.....	10
2.2	Bus Architecture.....	11
2.3	Signal Description.....	11
2.3.1	Power Connections.....	12

2.3.2	Clock Signals.....	12
2.3.3	System Control Signals.....	13
2.3.4	Slot Control Signals.....	15
2.3.5	DMA Control Signals.....	15
2.3.6	Addressing and Control Signals.....	17
 CHAPTER 3 BUS ARCHITECTURE		
3.1	Basic Zorro III Bus Cycles.....	19
3.1.1	Design Goals.....	20
3.1.2	Simple Bus Cycle Operation.....	20
3.2	Advanced Mode Support Logic.....	22
3.2.1	Bus Locking.....	22
3.2.2	Cache Support.....	22
3.3	Multiple Transfer Cycles.....	23
3.4	Quick Bus Arbitration.....	25
3.5	Quick Interrupts.....	27
3.6	Compatibility with Zorro II Devices.....	28
3.7	Zorro III Implementations.....	29
 CHAPTER 4 SIGNAL DESCRIPTION		
4.1	Power Connections.....	31
4.2	Clock Signals.....	32
4.3	System Control Signals.....	32
4.4	Slot Control Signals.....	34
4.5	DMA Control Signals.....	35
4.6	Address and Related Control Signals.....	35
4.7	Data and Related Control Signals.....	37
 CHAPTER 5 TIMING		
5.1	Standard Read Cycle Timing.....	40
5.2	Standard Write Cycle Timing.....	42
5.3	Multiple Transfer Cycle Timing.....	44
5.4	Quick Interrupt Cycle Timing.....	46
 CHAPTER 6 ELECTRICAL SPECIFICATIONS		
6.1	Expansion Bus Loading.....	49
6.1.1	Standard Signals.....	50
6.1.2	Clock Signals.....	50
6.1.3	Open Collector Signals.....	51
6.1.4	Non-bussed Signals.....	51
6.2	Slot Power Availability.....	51
6.3	Temperature Range.....	51

CHAPTER 7	MECHANICAL SPECIFICATIONS	
7.1	Basic Zorro III PIC.....	54
7.2	PIC with ISA Option.....	55
7.3	PIC with Video Option.....	56
CHAPTER 8	AUTOCONFIG®	
8.1	The AUTOCONFIG® Mechanism.....	57
8.2	Register Bit Assignments.....	58
CHAPTER 9	ZORRO III SIGNAL NAMES	
9.1	Physical and Logical Signal Names.....	63

PART II: THE AMIGA LOCAL BUS SLOT

CHAPTER 10	INTRODUCTION TO THE LOCAL BUS	
10.1	Intended Audience.....	67
10.2	Why a Local Bus Slot?.....	67
10.3	Why an Expansion Bus Slot?.....	68
CHAPTER 11	FUNCTIONALITY AND DESIGN GUIDELINES	
11.1	Slave Devices.....	69
11.2	Master Devices.....	70
11.2.1	Primary Bus Mastership.....	70
11.2.2	Secondary Bus Mastership.....	71
11.3	Clock Generation.....	72
11.4	Local Bus Design Criteria.....	73
CHAPTER 12	LOCAL BUS SIGNAL DESCRIPTIONS	
12.1	Power Connections	75
12.2	System Initialization.....	75
12.3	68030 Signals.....	76
12.4	Bus Arbitration Signals.....	78
12.5	Other Local Bus Signals.....	78
12.5	Clocks.....	79
12.5	Amiga 3000T Signals.....	79
12.5	Amiga 4000 Signals.....	80
12.6	Local Bus Signal Names and Pin Numbers	81

CHAPTER 13	LOCAL BUS FORM FACTORS	
13.1	A4000 Local Bus Form Factor (68040 Coprocessor Board).....	86
13.2	A3000 Local Bus Form Factor.....	87

PART III: THE AMIGA VIDEO SLOT

CHAPTER 14	THE AMIGA VIDEO SLOT	
14.1	Evolution of the Amiga Video Slot.....	89
14.2	Video Slot Connector 1.....	90
14.2.1	Power Connections.....	90
14.2.2	Clock Signals.....	91
14.2.3	Video Signals.....	92
14.2.4	Audio Signals.....	93
14.2.5	New A4000 Signals.....	93
14.3	Video Slot Connector 2.....	93
14.3.1	Additional Video Signals.....	93
14.3.2	Additional Audio Signals.....	94
14.3.3	Additional Grounding.....	95
14.3.4	Additional Clock Signals.....	95
14.3.5	Parallel Port Connections.....	95
14.4	Video Slot Pinout.....	97
 CHAPTER 15	 GENLOCK INTERFACE GUIDELINES	
15.1	Hardware Interpretation of HSY* and VSY * Reset Pulses.....	99
15.2	Pulse Duration.....	100
15.3	NTSC and PAL Genlock Interface Guidelines.....	100
 CHAPTER 16	 VIDEO BOARD FORM FACTORS	
16.1	A4000 Video Board Form Factor.....	104
16.2	A3000 Video Board Form Factor.....	105
 GLOSSARY		 107

TABLES AND FIGURES

Figure 1-1	A3000/A4000 Memory Map.....	4
Figure 2-1	A2000 vs. A3000/A4000 Bus Termination.....	10
Figure 2-2	Expansion Bus Clocks.....	13
Figure 2-3	Zorro II Bus Arbitration.....	16
Figure 3-1	Basic Zorro III Cycles.....	21
Figure 3-2	Multiple Transfer Read Cycles.....	24
Figure 3-3	Zorro III Bus Arbitration.....	26
Figure 3-4	Interrupt Vector Cycle.....	27
Figure 3-5	Zorro II Within Zorro III.....	28
Table 4-1	Memory Space Type Codes.....	36
Table 6-1	Zorro III Drive Types.....	50
Figure 8-1	Configuration Register Mapping.....	58

1

2

3

CHAPTER 1

INTRODUCTION

"Welcome, my son. Welcome to The Machine."

-Pink Floyd

This document describes the complete Zorro III bus, first implemented in the Amiga 3000 Computer. The Zorro III bus is a performance 32-bit expansion bus that is also upward compatible with the Zorro II bus (Amiga 2000 expansion bus). The main intent of the Zorro III bus is to allow fast 32-bit peripherals and memory devices to be added to a high performance Amiga, such as the Amiga 3000, while at the same time allowing standard Zorro II devices to be used wherever they make sense in such a system. This compatibility also insures that the Amiga 3000 will have a number of hardware and software compatible expansion devices available upon introduction, and that Amiga 2000 owners will be able to take their expansion card investment along with them should they migrate to a higher performance Amiga.

1.1 Intended Audience

This document was written primarily for hardware engineers interested in designing Plug In Cards (PICs) for the Zorro III expansion bus. While it may occasionally be of use to software engineers interfacing to such Zorro III PICs, Amiga system software provides an interface layer (*expansion.library* in the Amiga OS) which manages the needs of most card-level software. A reasonable level of microcomputer knowledge is prerequisite to get much meaning out of these pages. A good understanding of the Motorola 680x0 processors will be quite useful, as will be an understanding of the Zorro II expansion bus used on earlier Amiga computers such as the Amiga 2000.

1.2 Bug Reports

This is the second major publication of the *Zorro III Bus Specification*. While every effort has been made to keep it as accurate as possible, there is certainly the possibility that some errors have made it into this document. Anyone finding any error is encouraged to contact Commodore at the address below:

Dave Haynie/Amiga Systems Engineering
1200 Wilson Drive
West Chester, PA 19380

Bugs can also be reported on BIX or via Usenet; on BIX, use the "amiga.com/hardware" conference, or contact Dave Haynie directly as "hazy"; for Usenet users, bug reports can be sent to the address "{uunet,rutgers}!cbmvax!bugs" (use "cbmvax.cbm.commodore.com" if you like domain names); please also copy any such reports to "{uunet,rutgers}!cbmvax!daveh".

1.3 Amiga Bus History

The original Amiga computer, the Amiga 1000, was introduced in 1985. While it had no built-in standard for expandability, the capability for some form of expansion was considered extremely important; personal computer history up to that date had shown several times that an open hardware expansion capability was often critical to a personal computer's success and to its capability to adapt to new or unusual applications. The A1000 was designed with a connector giving access to the internal 68000 bus and a few other system signals. Shortly after introduction, the formal expansion specification for a card chassis that would connect to the A1000 was published. This bus became commonly known as the Zorro bus*. While the backplane specification was very easy to implement with 1985 PAL technology based on the existing 68000 signals, the specification did incorporate a number of advanced features. Far more sophisticated than the IBM-XT/AT and Apple II buses in common use at the time, the Zorro bus allowed any slot to master the bus, and it linked expansion cards with the system software. Addressing jumpers were eliminated, the card's address instead being assigned by software, and cards could easily be identified by software and linked with appropriate driver programs, all with a minimum of user intervention.

With the introduction of the Amiga 2000 system, the Zorro bus was changed slightly. Additional discrete interrupt lines were added, replacing the encoded lines that couldn't easily be used by any bus resident device. As it turns out, these additional encoded lines weren't any more useful, as they couldn't be disabled by software, and as such, they're no longer considered an official part of the Zorro II bus specification (they are supported as part of Zorro III). Finally, the form factor was changed to match that of the IBM PC-AT card, acting as both a cost reduction and allowing the Zorro II bus to offer the PC-AT bus as one optional secondary bus extension. This modified specification became commonly known as the Zorro II bus, and it's the

* The original "Zorro" name comes from the code name of one of the A1000 prototype boards. The "Zorro" board was the one that followed the "Lorraine", and was the board in the works when much of the expansion specifications were worked up. Since everyone uses the "Zorro" name, and no one's suggested a better name, I stick with it throughout this document.

Amiga bus standard that's been in use for most of the Amiga's life. And it's a bus standard that will continue to be important.

1.4 The Zorro III Rationale

With the creation of the Amiga 3000, it became clear that the Zorro II bus would not be adequate to support all of that system's needs. The Zorro II bus would continue to be quite useful, as the current Amiga expansion standard, and so it would have to be supported. A few unused pins on the Zorro II bus and the option of a bus controller custom LSI, gave rise to the Zorro III design, which supports the following features:

- Compatibility with all Zorro II devices.
- Full 32-bit address path for new devices.
- Full 32-bit data path for new devices.
- Bus speed independent of host system CPU speed.
- High speed bus block transfer mode.
- Bus locking for multiprocessor support.
- Cache disable for simple cache support.
- Fair arbitration for all bus masters.
- Cycle by cycle bus arbitration mode.
- High speed interrupt mode.

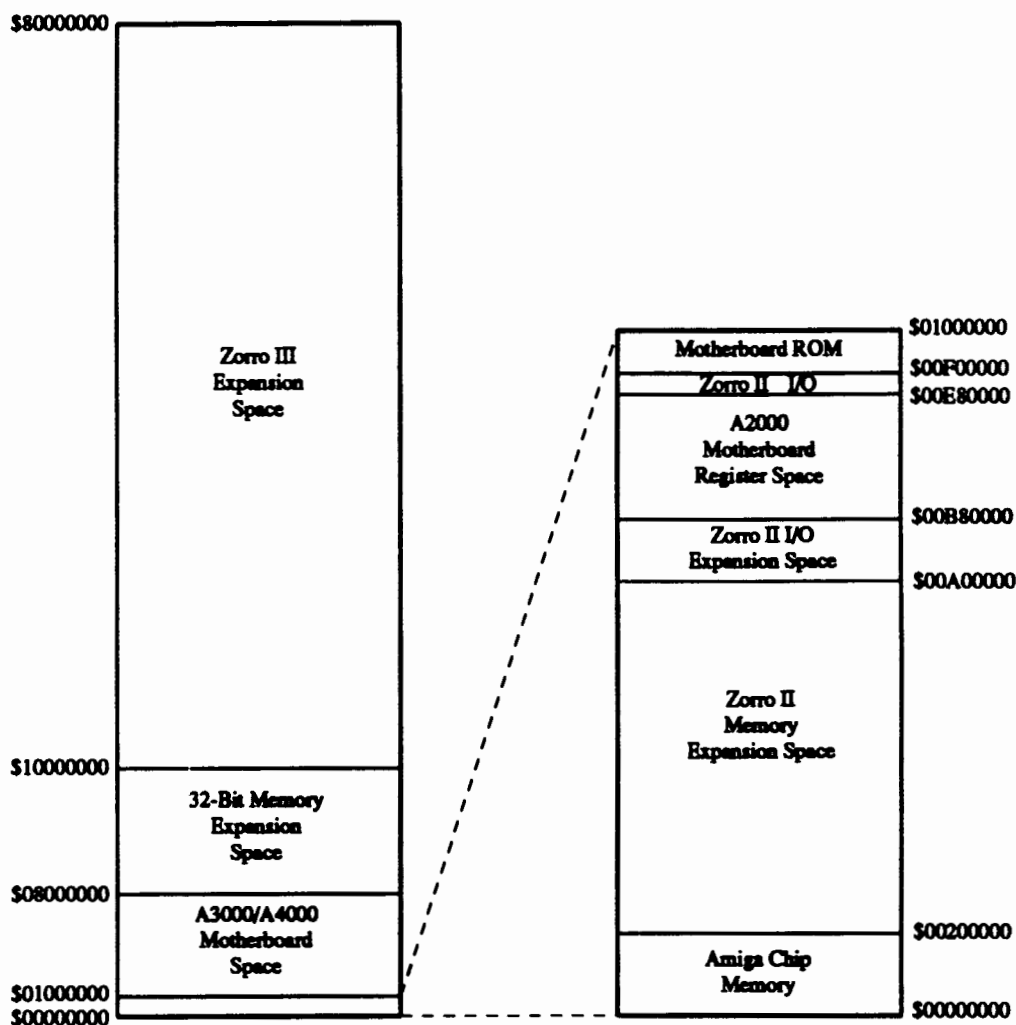
Some of the advanced features, such as burst modes, are designed in such a way as to make them optional; both master and slave arbitrate for them. In addition, it is possible with a bit of extra cleverness, to design a card that automatically configures itself for either Zorro II or Zorro III operation, depending on the status of a sensing pin on the bus.

The Zorro III bus is physically based on the same 100-pin single piece connector as the Zorro II bus. While some bus signals remain unchanged throughout bus operation, other signals change based on the specific bus mode in effect at any time. The bus is geographically mapped into three main sections, *Zorro II Memory Space*, *Zorro II I/O Space*, and *Zorro III Space*. The memory map in *Figure 1-1* shows how these three spaces are mapped in the A3000 and A4000 systems. The Zorro II space is limited to a 16 megabyte region, and since it has DMA access by convention to chip memory, it is in the original 68000 memory map for any bus implementation. The Zorro III space can physically be anywhere in 32-bit memory.

The Zorro III bus functions in one of two different major modes, depending on the memory address on the bus. All bus cycles start with a 32-bit address, since the full 32-bit address is required for proper cycle typing. If the address is determined to be in Zorro II space, a Zorro II compatible cycle is initiated, and all responding slave devices are expected to be Zorro II compatible 16-bit PICs. Should a Zorro III address be detected, the cycle completes when a Zorro III slave responds or the bus times out, as driven by the motherboard logic. It is very important that no Zorro III device respond in Zorro III mode to a Zorro II bus access; as the following chapters will reveal, the two types of cycles make very different use of many of the expansion bus lines, and serious buffer contention can result if the cycle types are somehow

mixed up. The Zorro III bus of course started with the Zorro II bus as its necessary base, but the Zorro III bus mechanisms were designed as much as possible to solve specific needs for high end Amiga systems, rather than extend any particular Zorro II philosophy when that philosophy no longer made any sense. There are actually several variations of the basic Zorro III cycle, though they all work on the same principles. The variations are for optimization of cycle times and for service of interrupt vectors. But all of this in due time.

Figure 1-1: A3000/A4000 Memory Map



1.5 Document Revision History

While there's significantly more real Zorro III hardware actually in existence at the time of this writing than when the first revision of this document was created, various Zorro III issues are still, from time to time, changing. In order to document these changes, this section was created. Although revision histories often discuss revisions in reverse chronological order, it's done here in chronological order to keep the subsection numbers consistent between revisions of this document.

1.5.1 Changes for Rev 0.90

The major changes in Rev 0.90 are actually additions. Specifically, the remaining parts of the Zorro III Timing (Chapter 5) and Mechanical (Chapter 7) specifications have been incorporated into this document. Additionally, the Zorro III design example in Appendix A.4 has been deleted. This simple and somewhat kludgy example has been supplanted by a more useful, straightforward, and thoroughly explained example, available as the separate document *BIGRAM 8/32: A Complete Zorro III Design Example*. In general, we expect both documents to be distributed together, but as always, CATS can assist in the procurement of any missing information.

1.5.2 Changes for Rev 0.91

In the Introduction (Chapter 1), the official revision history has been added as a standard part of this document. The Zorro III Bus Architecture (Chapter 3), section 3.5, has been changed to reflect the revised Quick Interrupt vector allocation mechanism. In the Timing specification (Chapter 5), corrections have been made: timing parameter 6 was left out of the section 5.3 timing, and timing parameter 19 was incorrectly specified in section 5.4. In the AUTOCONFIG® specification (Chapter 8), corrections have been made to the addressing tables for registers 44 and 48. Also the Quick Interrupt Enable bit (register 08:4) and Vector Register (register 50) have been deleted from the specification. Quick Interrupt Vector allocation is now handled via an Exec call, a single configuration unit can have several vectors, and the means of storage on a PIC is up to the designer.

1.5.3 Changes for Rev 1.00

In the AUTOCONFIG® specification (Chapter 8), bit 4 of register 08 has been changed to always read 1 for Zorro III PICs. This change was necessary for compatibility with 1.3, due to a bug in the 1.3 expansion.library. Also, the nybble write configuration mode for the Zorro III configuration block has been eliminated, only byte and word writes are now supported.

1.5.4 Changes for Rev 1.01

The AUTOCONFIG® specification change listed in 1.5.3 was missing from Chapter 8 in Rev 1.00 of the spec, now it's actually there. Additionally, some clarification on the proper action of slave cards and bus error conditions has been added to Chapter 4.

1.5.5 Changes for 1.10

The /INT₁, /INT₄, /INT₅, and /INT₇ lines have been eliminated from the Zorro III bus specification. Although the A3000 hardware supports these, some AmigaOS software conventions makes their use impossible under the AmigaOS. These lines are now considered reserved and are not present at all in the A4000. Also, in section 3.5, the vector poll command code was given as 16, where it's actually 15; this has been corrected.

1.5.6 Changes for Rev 1.11

Revision 1.11 of this document includes new information about the A4000 and has additional chapters on the Local Bus (Coprocessor) connector and the video connectors present in all A4000 and A3000 computers.

CHAPTER 2

ZORRO II COMPATIBILITY

"In Jersey anything's legal, as long as ya don't get caught."

- Traveling Wilburys

The Zorro III bus is a rather extensive superset of the Zorro II bus design. The compatibility is based on distinct bus modes, rather than a simple extension to the existing bus mechanisms. Through the use of an integrated bus controller (the Fat Buster chip), the expansion bus configures itself differently for the 16-bit A2000-compatible Zorro II modes than the 32-bit Zorro III modes.

As a result, while there are still only 100 pins on the expansion bus, some pins change function considerably depending on the bus activity that's currently in progress. While the Zorro II modes of the Zorro III bus are as compatible as possible with the Zorro II bus specification (especially the A2000 implementation of this specification), there are some small differences between the two expansion buses.

Aside from these differences, in general, it's important to understand the Zorro II bus in order to understand the Zorro III bus. The general features of Zorro III, like autoconfiguration, the master-slave bus architecture, and the physical attributes come from the Zorro II expansion bus. Other features of the Zorro III bus address shortcomings of the Zorro II architecture, but Zorro II has a hand in how some of these shortcomings are solved under Zorro III. Those with a full understanding of the Zorro II bus will mainly be concerned with the possible bus incompatibilities listed here.

2.1 Changes From The A2000 (Zorro II) Bus

While much effort has been made to assure that the Zorro II mode of the Zorro III bus is as compatible as possible with the A2000, there are a few points to consider here. Primarily, the Zorro II modes of the Zorro III bus are driven with a state machine that emulates the 68000 bus

protocol. This emulation must be based on the published Motorola specifications detailing 68000 bus behavior. While this has the interesting effect of changing the Zorro II bus from CPU-dependent to CPU-independent, there's some margin for error. Zorro II PICs also designed to these specifications should have no trouble in the A3000 bus in most cases. However, anything designed based on observed 68000 behavior rather than documented 68000 operation is at serious risk of failing in an A3000 or A4000, as one might expect. There are also actual documented differences, which are listed below.

2.1.1 6800 Bus Interface

A major difference between the Zorro II mode of Zorro III and the 'real' Zorro II bus are the absence of the signals /VPA and /VMA, which comprise the 6800/6502 peripheral support mechanism that's part of the 68000 bus interface. This mechanism was never a supported part of the Zorro II specification, however, and it should not be used by any PIC. Any Zorro II PIC that depends on /VPA or /VMA will not work in the A3000 bus. It was, in fact, impossible to legally use this on the A2000 bus. The E clock is, however, supported on the Zorro III bus, though its duty cycle may vary in some situations.

2.1.2 Bus Memory Mapping and Cache Support

Another change to the Zorro II implementation is that the bus mapping logic works a little differently. Zorro II address space is broken up into memory and I/O address space. Memory space is the standard 8 megabyte space from \$00200000-\$009FFFFFF. The I/O address space is mapped at \$00E80000-\$00FFFFFF, and a new 1.5 megabyte section (previously reserved for motherboard devices) from \$00A00000-\$00B7FFFF. Zorro II cycles are not generated for non-Zorro II address space, even for 68000 space resources on the local bus. So, for example, a CPU access to chip memory would be visible to a Zorro II PIC in an A2000 backplane, but invisible to that same PIC in an A3000 backplane. Since this extra information on the Zorro II backplane can't be legally used by any PIC anyway, it should not be used by any existing A2000 PICs.

The reason for the two distinct mapping regions is for cache support of Zorro II PICs. All access by the local bus* master to Zorro II memory space results in the local bus cache enable signal being driven and a full port read (eg, both bytes) regardless of the actual data transfer size being requested. A local bus access to Zorro II I/O space results in the local bus cache disable signal being driven and the data strobes for reads indicating the requested transfer size. This cache mapping mechanism was first implemented in the A2630 coprocessor card, so it's not an entirely new concept.

* The local bus, motherboard bus, and CPU bus are the same thing; the immediate 680x0 bus connected directly to the CPU in an Amiga computer. Current Amiga computers typically support three distinct buses; the expansion bus, local bus, and chip bus. From the point of view of the expansion bus, the local and chip buses appear as a unified device which may be master or slave to the expansion bus.

2.1.3 Bus Synchronization Delays

Due to the asynchronous nature of the local-to-expansion bus interface for Zorro II cycles, extra wait states may occasionally be added for local to expansion or expansion to local cycles. These are generally manifested as delays between consecutive cycles, since the bus controller is not going to require extra waiting during the cycle -- things will have already been synchronized at that point. The synchronization problems get more difficult for Zorro II master access to local bus slaves, and as a result, wait states here are very common. The actual number of wait states generated in any case will be based on the particular implementation.

2.1.4 Zorro II Master Access to Local Slaves

The only supported local bus resource that's guaranteed accessible to a Zorro II expansion bus master as a slave device is chip bus memory. All I/O devices are implementation dependent and not supportable via DMA. Any attempted access to unsupported local bus resources as expansion slaves will result in an error condition being signalled on both the local and the expansion buses. Most other local bus resources, such as local bus fast memory, are located outside of Zorro II space on most systems and obviously not available to Zorro II masters.

2.1.5 Bus Arbitration and Fairness

The Zorro II bus is now arbitrated fairly. The normal slot-based order of precedence is given to requesting devices, just as in the A2000 implementation. As always, once a bus master assumes bus mastership, it has the bus for as long as it wants the bus (of course, trouble can result if a device takes the bus over for too long). Once a master gives up the bus, it will not be granted it back until all subsequent requests have been serviced. Bus arbitration at its best will be slightly slower than in the A2000 implementation, due to the fairness logic, but it is impossible to jam the arbiter with asynchronous bus requests as in the A2000. The new style arbiter also holds off bus grants while hidden local bus cycles are in progress, so there's no guarantee of a minimum time between bus request and bus grant specified.

2.1.6 Intelligent Cycle Spacing

In order to permit a free intermix of Zorro II and Zorro III cycles, the bus control logic is capable of making intelligent decisions when spacing bus cycles. In some cases, a Zorro II cycle has some component that would naturally extend into a following cycle. The cycle spacing logic detects such a condition, and refuses to start a new cycle until the current one is complete, even if this extends beyond the defined bounds of a Zorro II cycle. For Zorro II PICs that really follow the Zorro II specifications, this should have no effect. However, any Zorro II PIC that holds signals much beyond the end of a cycle, especially critical signals like /SLAVE and /DTACK, will likely incur additional wait states on the Zorro III bus. This is not intended as a license for making sloppy expansion card designs, just an acknowledgement that some Zorro II devices may cause a conflict with the faster Zorro III bus timings, and the best thing to do about such cases is to make them work, even with a possible performance penalty.

2.1.7 Bus Drive and Termination

Finally, the Zorro III bus uses different bus termination than that in the A2000. The Zorro II specification didn't specify the termination expected; backplanes were built that didn't even have termination. The A2000 bus used a circuit consisting of a capacitor in series with a resistor to ground for most of the bus signals. This has good reflection cancelling properties without increasing crosstalk (a major concern on the 2-layer A2000 motherboard), but it does slow things down measureably. The main reason for the change on the A3000 backplane is to support the faster Zorro III bus modes. The multi-layer A3000 motherboard permits a

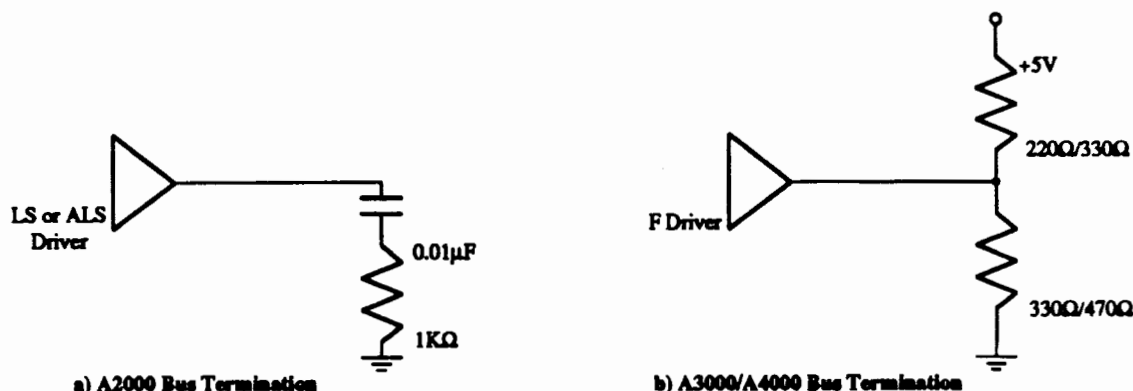


Figure 2-1: A2000 vs. A3000/A4000 Bus Termination

reasonably high current bus without undue crosstalk. The thevenin termination makes switching logic levels start from a midpoint instead of a rail, especially for a bus coming out of tri-state (which, based on the Zorro III design, happens constantly). This should not cause problems with Zorro II cards, but it's conceivable that some cards may need to be adjusted to work in this bus (the Zorro III bus requires somewhat higher current capability than the Zorro II bus does. The A3000 and A4000 do not support enough slots for loading to be a likely problem, but future Zorro III backplanes will have more slots and make this an important consideration).

2.1.8 DMA Latency and Overlap

Zorro II bus masters in a Zorro III backplane will, in many cases, receive a bus grant much sooner than they would in a standard Zorro II backplane. Additionally, in some cases, expansion bus cycles will overlap local bus cycles. The latency incurred on the Zorro II bus during heavy custom chip activity has been greatly reduced for any Zorro III bus master. This should be transparent to the card in question, though it's a good thing to be aware of.

2.1.9 Power Supply Differences

The Zorro II bus is defined as supplying +5VDC @ 2 Amps to each slot, with one slot per backplane supplying 5.0VDC @ 4.0 Amps. The Zorro III bus only provides the 5.0VDC @ 2.0 Amps for each slot.

2.2 Bus Architecture

The Zorro II bus is a simple extension of the 68000 processor bus. Those without a good knowledge of the 68000 local bus will find *The 68000 User's Manual* from Motorola an excellent reference for many Zorro II issues. The *A500/A2000 Technical Reference Manual* from Commodore-Amiga is also required reading for any Zorro II design issues, as it includes a complete description of all the Commodore-Amiga details that aren't part of the 68000 specification.

The basic Zorro II bus is a buffered version of the 68000 processor bus, physically provided on a 100-pin one-piece connector. The bus is 16 bits wide, and provides 24 bits of addressing information. A bus cycle looks exactly like a 68000 bus cycle. The cycle is defined by an address strobe, terminated by a data transfer strobe, and qualified by a read/write strobe, some memory space qualifiers, and one or two byte selection strobes. The basic bus cycle runs for a total of four cycles of a 7.16 MHz clock, though it can be extended to add wait states when required.

The Zorro II bus adds a number of features to the basic 68000 CPU bus. It supplies some Amiga system signals that are useful for expansion card designs, such as many of the Amiga system clocks. The bus provides a default data transfer signal, which expansion cards can easily use and modify rather than go to the trouble of creating their own. It provides a number of discrete interrupt lines which are mixed to provide the 68000 with its standard encoded interrupts. The 68000 bus arbitration protocol is used to allow multiple bus masters; arbitration of the bus requests are managed by the Zorro II bus controller to avoid contention between multiple masters. And of course the bus supplies a number of supply voltages for powering cards.

A powerful aspect of the Zorro II bus is its convention for automatically configuring expansion cards, AUTOCONFIG®. On system powerup, the system software interrogates each board to determine what kind of board is installed and how much memory space it needs on the bus. The software then tells each board where to reside in memory. The bus provides hardware lines to allow the boards to be configured in a daisy chained fashion regardless of which slots they occupy and to prevent damage to boards if accidentally configured to reside at the same memory location. Firmware standards also permit software to autoboot or autoinitialize any board, to match soft-loaded device drivers with individual boards, and to link memory boards into the appropriate system memory lists.

2.3 Signal Description

The Zorro II bus can be broken down into various logical signal groups. Some of these groups are unchanged in the Zorro III bus modes, others are drastically different. This section makes note of the original Zorro II name for each signal and the current Zorro III physical pin name for each signal, where different. Some of this information will be repeated in the Zorro III chapters, where appropriate; nothing in this chapter is considered critical to understanding the Zorro III bus, but it is useful. As previously mentioned, the A2000 bus signals unsupported by

the Zorro II specification have been deleted from both the Zorro III specification and its implementation in the A3000 and A4000; this section will, however, document those signals for reference purposes. Please see Chapter 9 for a complete list with pin numbers of the various logical signals that appear on the physical bus during the different phases of the Zorro II and Zorro III bus cycles.

2.3.1 Power Connections

The Zorro III expansion bus provides several different voltages designed to supply expansion devices. There are no changes here that affect Zorro II cards.

Digital Ground (Ground)

This is the digital supply ground used by all expansion cards as the return path for all expansion supplies.

Main Supply (+5VDC)

This is the main power supply for all expansion cards, and it is capable of sourcing large currents; each expansion slot can draw up to 2.0 Amps @ +5VDC. The extra power for one card in any backplane drawing up to 4.0 Amps @ +5VDC is no longer supported.

Negative Supply (-5VDC)

This is a negative version of the main supply, for small current loads only. There is no maximum load specified for the Zorro II bus on a per-slot basis; the A2000 implementation specifies 0.3 Amps @ -5VDC for the entire system.

High Voltage Supply (+12VDC)

This is a higher voltage supply, useful for communications cards and other devices requiring greater than digital voltage levels. This is intended for relatively small current loads only. There is no maximum load specified for the Zorro II bus on a per-slot basis; the A2000 implementation specifies 8.0 Amps @ +12VDC for the entire system, most of which is normally devoted to floppy and hard disk drive motors, not slots.

Negative High Supply (-12VDC)

Negative version of the high voltage supply, also commonly used in communications applications, and similarly intended for small loads only. There is no maximum load specified for the Zorro II bus on a per-slot basis; the A2000 implementation specifies 0.3 Amps @ -12VDC for the entire.

2.3.2 Clock Signals

The Zorro III expansion bus provides clock signals for expansion boards. These clocks are for synchronous Zorro II designs and for other synchronous activity such as bus arbitration. While originally based on Amiga local bus clocks, these have no guaranteed relationship to any local bus activity in newer Amiga computers, but are maintained in Amiga computers as part of the expansion bus specification. The relationship between these clocks is illustrated in *Figure 2-2*.

/C1 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the falling edge of the 7M system clock.

/C3 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the rising edge of the 7M system clock.

CDAC Clock

This is a 7.16 MHz system clock (7.09 MHz on PAL systems) which trails the 7M clock by 90° (approximately 35ns).

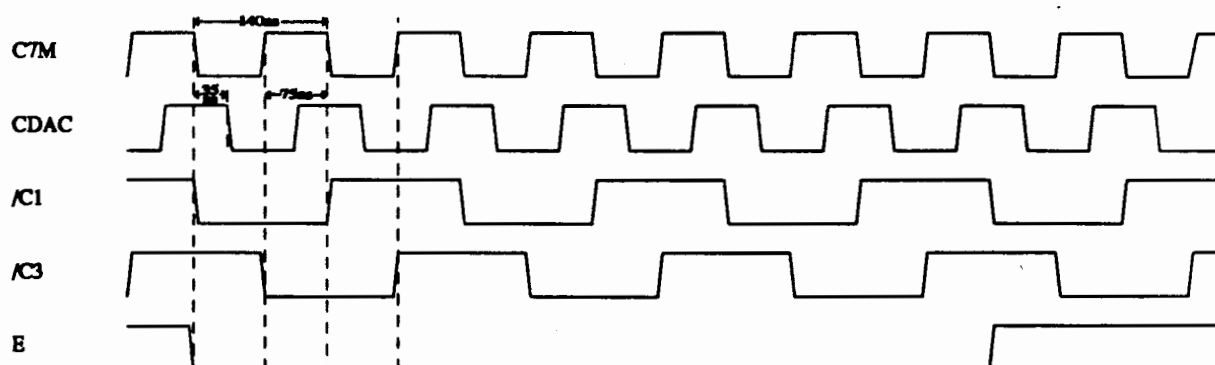


Figure 2-2: Expansion Bus Clocks

E Clock

This is the 68000 generated "E" clock, used for 6800 family peripherals driven by "E" and 6502 peripherals driven by Φ_2 . This clock is four 7M clocks high, six clocks low, as per the 68000 spec. Note that the bus does not support the rest of the 68000's 6800/6502 compatible interface; there may be better ways to clock such devices.

7M Clock

This is the 7.16 MHz system clock (7.09 MHz on PAL systems). This clock forms the basis for all Zorro II/68000 compatible activity, and for various other system functions, such as bus arbitration.

2.3.3 System Control Signals

The signals in this group are available for various types of system control; most of these have an immediate or near immediate effect on expansion cards and/or the system CPU itself.

Bus Error (/BERR)

This is a general indicator of a bus fault condition. Any expansion card capable of detecting a hardware error relating directly to that card can assert /BERR when that bus error condition is detected, especially any sort of harmful hardware error condition. This

signal is the strongest possible indicator of a bad situation, as it causes all PICs to get off the bus, and will usually generate a level 2 exception on the host CPU. For any condition that can be handled in software and doesn't pose an immediate threat to hardware, notification via a standard processor interrupt is the better choice. The bus controller will drive /BERR in the event of a detected bus collision or DMA error (an attempt by a bus master to access local bus resources it doesn't have valid access permission for).

All cards must monitor /BERR and be prepared to tri-state all of their on-bus output buffers whenever this signal is asserted. The current bus master should, if possible, retry the bus cycle after /BERR is negated unless conditions warrant otherwise. Since any number of devices may assert /BERR, and all bus cards must monitor it, any device that drives /BERR must drive with an open collector or similar device capable of sinking at least 12ma, and any device that monitors /BERR should place a minimal load on it (1 "F" type load or less). This signal is pulled high by a passive backplane resistor.

System Reset (/RST, /BUSRST) \equiv (/RESET, /IORST) for Zorro III

The bus supplies two versions of the system reset signal. The /RST signal is bidirectional and unbuffered, allowing an expansion card to hard reset the system. It should only be used by boards that need this reset capability, and is driven only by an open collector or similar device. The /BUSRST signal is a buffered output-only version of the reset signal that should be used as the normal reset input to boards not concerned with resetting the system on their own. All expansion devices are required to reset their autoconfiguration logic when /BUSRST is asserted. This signal is pulled high by a passive backplane resistor.

System Halt (/HLT)

This signal is similar to the 68000 processor halt signal, and is driven by a PIC with an open-collector or similar gate only. Its main use is to indicate a full-system reset. Based on the 68000 conventions, an I/O-only reset, such as initiated by the 680x0 RESET instruction, will drive only /RST and /BUSRST on the bus. A full-system reset, such as a powerup reset or a keyboard reset, drives /HLT low as well. PICs that wish to reset the system CPU as well as the bus and I/O devices drive /RST and /HLT, some bus devices such as processor cards may internally reset only on full-system resets. This signal is pulled high by a passive backplane resistor.

System Interrupts

Six of the decoded, level sensitive 680x0 interrupt inputs were originally available on the expansion bus, and these are labelled as /INT₂, /INT₆, /EINT₁, /EINT₄, /EINT₅, /EINT₇ on the Zorro II bus. Only the /INT₂ and /INT₆ interrupt inputs are actually supported by Commodore-Amiga as part of the Zorro II specification; the A2000 hardware did not provide the to software the required support mechanisms for the safe use of these lines. Each of these interrupt lines are shared by wired ORing, thus each line must be driven by an open-collector or equivalent output type, and all are pulled high by passive backplane resistors.

2.3.4 Slot Control Signals

This group of signals is responsible for the control of things that happen between expansion slots.

Slave (/SLAVEN)

Each slot has its own /SLAVE output, driven actively, all of which go into the collision detect circuitry. The "N" refers to the expansion slot number of the particular /SLAVE signal. Whenever a Zorro II PIC is responding to an address on the bus, it must assert its /SLAVE output within 35ns of /AS asserted. The /SLAVE output must be negated at the end of a cycle within 50ns of /AS negated. Late /SLAVE assertion on a Zorro II bus can result in loss of data setup times and other problems. A late /SLAVE negation for Zorro II cards can cause a collision to be detected on the following cycle. While the Zorro III sloppy cycle logic eliminates this fatal condition, late /SLAVE negation can nonetheless slow system performance unnecessarily. If more than one /SLAVE output occurs for the same address, or if a PIC asserts its /SLAVE output for an address reserved by the local bus, a collision is registered and results in /BERR being asserted.

Configuration Chain (/CFGIN_N, /CFGOUT_N)

The slot configuration mechanism uses the bus signals /CFGOUT_N and /CFGIN_N, where "N" refers to the expansion slot number. Each slot has its own version of each, which make up the configuration chain between slots. Each subsequent /CFGIN is a result of all previous /CFGOUTs, going from slot 0 to the last slot on the expansion bus. During the AUTOCONFIG® process, an unconfigured Zorro PIC responds to the 64K address space starting at \$00E80000 if its /CFGIN signal is asserted. All unconfigured PICs start up with /CFGOUT negated. When configured, or told to "shut up", a PIC will assert its /CFGOUT, which results in the /CFGIN of the next slot being asserted. The backplane passes on the state of the previous /CFGOUT to the next /CFGIN for any slot not occupied by a PIC, so there's no need to sequentially populate the expansion bus slots.

Data Output Enable (DOE)

This signal is used by an expansion card to enable the buffers on the data bus. The main Zorro II use of this line is to keep PICs from driving data on the bus until any other device is completely off the bus and the bus buffers are pointing in the correct direction. This prevents any contention on the data bus.

2.3.5 DMA Control Signals

There are various signals on the expansion bus that coordinate the arbitration of bus masters. Native Zorro III bus masters use some of the same logical signals, but their arbitration protocol is considerably different.

PIC is DMA Owner (/OWN)

This signal is asserted by an expansion bus DMA device when it becomes bus master. This output is to be treated as a wired-OR output between all expansion slots, any of

which may have a PIC signalling bus mastership. Thus, this should be driven with an open-collector or similar output by any PIC using it. This signal is the main basis for data direction calculations between the local and expansion busses, and is pulled up by a backplane resistor.

Slot Specific Bus Arbitration (/BR_N, /BG_N)

These are the slot-specific /BR_N and /BG_N signals, where "N" refers to the expansion slot number. The bus request from each board is taken in by the bus controller and ultimately used to take over the system from 680x0 on the local bus. The bus controller eventually returns one bus grant to the winner among all requesting PICs. From the point of view of

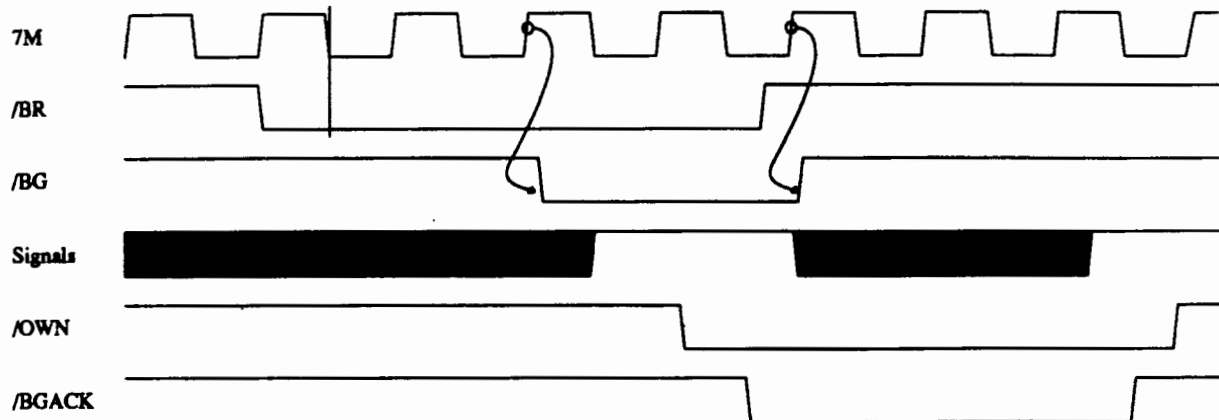


Figure 2-3: Zorro II Bus Arbitration

the individual PIC, the protocol is very similar to that of the 68000 arbitration mechanism. The PIC asserts /BR_N on the rising edge of 7M; some time later, /BG_N is returned on the falling edge of 7M. The PIC waits for all bus activity to finish, asserts /OWN followed by /BGACK, then negates /BR_N, assuming bus mastership. It retains mastership until it negates /BGACK followed by /OWN.

Bus Grant Acknowledge (/BGACK)

Any Zorro II PIC that receives a bus grant asserts this signal as long as it maintains bus mastery. This signal may never be asserted until the bus grant has been received, /AS is negated, /DTACK is negated, and /BGACK itself is negated, indicating that all other potential bus masters have relinquished the bus. This output is driven as a wired-OR output, so all PICs must drive it with an open collector or equivalent device, and a passive pullup is supplied by the backplane.

Bus Want/Clear (/GBG) \equiv (/BCLR) for Zorro III

This signal is asserted by the bus controller to indicate that a PIC wants to master the bus. A bus master assumes that the host CPU wants the bus, and that any time wasted as master is stealing time from the CPU. To avoid such waste, a master should use cache or FIFO to grab slow-coming data, and then transfer it all at once. /BCLR is asserted to indicate that additionally, another PIC wants the bus, and the current bus master should get off as soon as possible. This signal is equivalent to /GBG on the A2000 bus.

2.3.6 Addressing and Control Signals

These signals are various items used for the addressing of devices in Zorro II mode by the local bus and any expansion DMA devices. Most of these signals are very much like 68000 generated bus signals bi-directionally buffered to allow any DMA device on the bus to drive the local bus when such a device is the bus master.

Read Enable (READ)

This is the read enable for the bus, which is equivalent to the 68000's R/W output. READ asserted during a bus cycle indicates a read cycle, READ negated indicates a write cycle. Note that this signal may become valid in a cycle earlier than a 68000 R/W line would, but it remains valid at least as long at the cycle's end.

Address Bus (A₁-A₂₃)

This is logically equivalent to the 68000's address bus, providing 16 megabytes of address space, although much of that space is not assigned to the expansion bus (see the memory map in *Figure 1-1*).

Address Strobe (/AS) \equiv (/CCS) for Zorro III

This is equivalent to the 68000 /AS, called /CCS, for Compatibility Cycle Strobe, in the Zorro III nomenclature. The falling edge of this strobe indicates that addresses are valid, the READ line is valid, and a Zorro II cycle is starting. The rising edge signals the end of a Zorro II bus cycle, signaling the current slave to negate all slave-driven signals as quickly as possible. Note that /CCS, like /AS, can stay asserted during a read-modify-write access over multiple cycle boundaries. To correctly support such cycles, a device must consider both the state of /CCS and the state of the data strobes. Many current Zorro II cards don't correctly support this 680x0 style bus lock.

Data Bus (D₀-D₁₅)

This is a buffered version of the 680x0 data bus, providing 16 bits of data accessible by word or either byte. A PIC uses the DOE signal to determine when the bus is to be driven on reads, and the data strobes to determine when data is valid on writes.

Data Strokes (/UDS, /LDS) \equiv (/DS₃, /DS₂) for Zorro III

These strobes fall on data valid during writes, and indicate byte select for both reads and writes. The lower strobe is used for the lower byte (even byte), the upper strobe is used for the upper byte (odd byte). There is one slight difference between these lines and the 68000 data strobes. On reads of Zorro II memory space, both /DS₃ and /DS₂ will be asserted, no matter what the actual size of the requested transfer is. This is required to support caching of the Zorro II memory space. For Zorro II I/O space, these strobes indicate the actual, requested byte enables, just as would a 68000 bus master.

Data Transfer Acknowledge (/DTACK)

This signal is used to normally terminate both Zorro bus cycles. For Zorro II modes, it is equivalent to the 68000's Data Transfer Acknowledge input. It can be asserted by the

bus slave during a Zorro II cycle at any time, but won't be sampled by the bus master until the falling edge of the S4 state on the bus. Data will subsequently be latched on the S6 falling edge after this, and the cycle terminated with /AS negated during S7. If a Zorro II slave does nothing, this /DTACK will be driven by the bus controller with no wait states, making the bus essentially a 4 cycle synchronous bus. Any slow device on the bus that needs wait states has two options. It can modify the automatic /DTACK negating XRDY to hold off /DTACK. Alternately, it may assert /OVR to inhibit the bus controller's generation of /DTACK, allowing the slave to create its own /DTACK. Any /DTACK supplied by a slave must be driven with an open-collector or similar type output; the backplane provides a passive pullup.

Processor Status (FC0-FC2)

These signals are the cycle type or memory space bits, equivalent for the most part with the 68000 Processor Status outputs. They function mainly as extensions to the bus address, indicating which type of access is taking place. For Zorro II devices, any use of these lines must be gated with /BGACK, since they are not driven valid by Zorro II bus masters. However, when operating on the Zorro III backplane, Zorro II masters that don't drive the function codes will be seen generating an FC1 = 0, which results in a valid memory access. Zorro II cycles are not generated for invalid memory spaces when the CPU is the bus master.

/DTACK Override (/OVR)

This signal is driven by a Zorro II slave to allow that slave to prevent the bus controller's /DTACK generation. This allows the slave to generate its own /DTACK. The previous use of this line to disable motherboard memory mapping, which was unsupported on the A2000 expansion bus, has now been completely removed. The use of XDRY or /OVR in combination with /DTACK is completely up to the board designer -- both methods are equally valid ways for a slave to delay /DTACK. In Zorro III mode, this pin is used for something completely different.

External Ready (XRDY)

This active high signal allows a slave to delay the bus controller's assertion of /DTACK, in order to add wait states. XRDY must be negated within 60ns of the bus master's assertion of /AS, and it will remain negated until the slave wants /DTACK. The /DTACK signal will be asserted by the bus controller shortly following the assertion of XRDY, providing the bus cycle is a S4 or later. XRDY is a wired-OR from all PICs, and as such, must be driven by an open collector or equivalent output. In Zorro III mode, this pin is used for something completely different.

CHAPTER 3

BUS ARCHITECTURE

"We follow in the steps of our ancestry, and that cannot be broken."

-Midnight Oil

While the Zorro II bus design was based in a large part on an already existing bus cycle, the 68000 cycle, the Zorro III bus design had a much different set of preconditions. It is not modeled after any particular CPU specific bus protocol, but instead it's a logical outgrowth of both the need to support Zorro II cards on the same bus and the need to achieve various modern feature and performance goals. These goals were summarized in Chapter 1, now they'll be covered in greater detail here.

3.1 Basic Zorro III Bus Cycles

The basic Zorro III bus cycle is a multiplexed address/data cycle which supplies a full 32 bits worth of address and data per simple cycle. The cycle is a fully asynchronous cycle. The bus master for a given cycle supplies strobes to indicate when address is valid, write data is valid, and read data may be driven. In return, the bus slave for a cycle supplies a strobe to indicate that it is responding to a bus address, and a strobe to indicate that it is done with the bus data for a write cycle, or has supplied valid bus data for a read cycle. The minimum theoretical bus speed is governed only by setup and hold time requirements for the various bus signals. Actual bus speeds is always a function of the bus master and bus slave active for a given cycle. This is considerably different than the way things work under the Zorro II bus, and for several good reasons, which are explained below.

3.1.1 Design Goals

For any computer bus, there are two basic possibilities concerning the fundamental operation of the bus; it's either synchronous or asynchronous. The difference is simple -- the synchronous bus is ultimately tied to a clock of some sort, while the asynchronous bus has no defined relationship to any clock signal. While Motorola specifies the 68000 bus cycle as an asynchronous cycle, they're really referring to the fact that most 68000 inputs are internally synchronized with the bus clock, and therefore, synchronous setup times on the bus do not have to be met to avoid metastability. But the 68000 bus, and the Zorro II bus by extension, are synchronous buses, based on a single bus clock (called E7M on the Zorro II bus). Most Zorro II signals are asserted relative to an edge of the bus clock, and most Zorro II inputs are sampled on an edge of the bus clock. The minimum Zorro II cycle is four bus clocks long, and every wait state added, regardless of the method, will result in a single additional bus clock wait, regardless of the asynchronous appearance of the termination and wait signals on the Zorro II bus.

The Zorro III bus is a fully asynchronous bus, in that all bus events are driven by strobes, and there is no reference clock. The choice of an asynchronous versus a synchronous bus design is governed by the intended application of the bus. Synchronous designs are preferred when a CPU and a memory system (e.g., master and slave) can be very tightly coupled to each other. Such designs generally require a tight adherence to timing based on the specific CPU. This is optimal for tightly coupled systems, such as fast memory on the A3000 local bus. Synchronous designs can also be easier to do accurately, as the designer can use clock edges for scheduling events, and there's never any need to waste time in synchronizers to achieve a reliable design.

The design goals for an expansion bus are considerably different. While a fast memory circuit on a system motherboard can change for every new and better design, it's not feasible to require redesign of any significant number of expansion cards every time an improved motherboard design is created. And while a synchronous transfer can be optimal for matched clocks, it can be very inefficient for mismatched CPU and expansion clocks, as synchronizer delays must be introduced for any reliable operation. The A3000 project started with the need to support CPU systems at 16MHz and at 25MHz, and it's obvious that the growth of CPU clock speed will be here for some time to come. Zorro III cards are based on asynchronous handshaking between master and slave in both directions. This means that, as long as masters and slaves manage their own needs, any slave can work with any master. But as masters and slaves improve with technology, bus transfer speeds can automatically increase, without rendering any slower cards obsolete. The Zorro III bus attempts to address the needs of device expansion as much as the needs of memory expansion.

3.1.2 Simple Bus Cycle Operation

The normal Zorro III bus cycle is quite different than the Zorro II bus in many respects. *Figure 3-1* shows the basic cycle. There is no bus clock visible on the expansion bus; the standard Zorro II clocks are still active during Zorro III cycles, but they have no relationship to the Zorro II bus cycle. Every bus event is based on a relationship to a particular bus strobe, and strobes are alternately supplied by master and slave.

A Zorro III cycle begins when the bus master simultaneously drives addressing information on the address bus and memory space codes on the FC_N lines, quickly following that with the assertion of the Full Cycle Strobe, /FCS; this is called the *address phase* of the bus. Any active slaves will latch the bus address on the falling edge of /FCS, and the bus master will tri-state the addressing information very shortly after /FCS is asserted. It's necessary only to latch A₃₁-A₈; the low order A₇-A₂ addresses and FC_N codes are non-multiplexed.

As quickly as possible after /FCS is asserted, a slave device will respond to the bus address by asserting its /SLAVE_N line, and possibly other special-purpose signals. The autoconfiguration process assigns a unique address range to each PIC base on its needs, just as on the Zorro II bus. Only one slave may respond to any given bus address; the bus controller will generate a /BERR signal if more than one slave responds to an address, or if a single slave

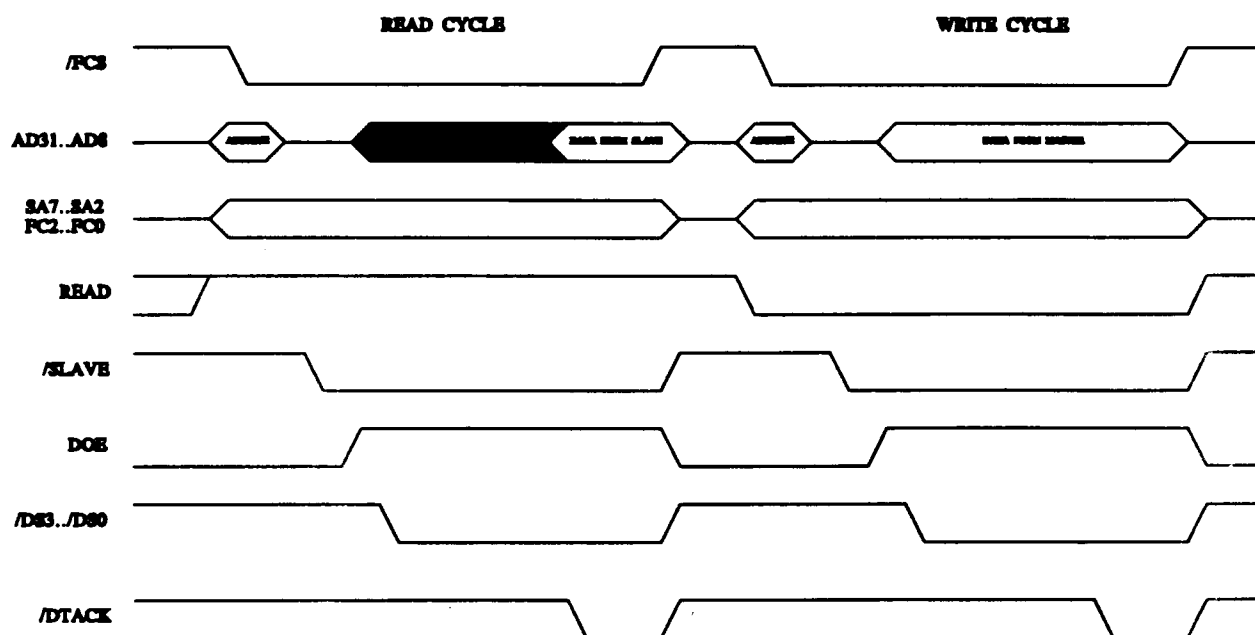


Figure 3-1: Basic Zorro III Cycles

responds to an address reserved for the local bus (this is called a bus collision, and should never happen in normal operation). Slaves don't usually respond to CPU memory space or other reserved memory space types, as indicated by the memory space code on the FC_N lines (see Chapter 4 for details)!

The *data phase* is the next part of the cycle, and it's started when the bus master asserts DOE onto the bus, indicating that data operations can be started. The strobes are the same for both read and write cycles, but of course the data transfer direction is different.

For a read cycle, the bus master drives at least one of the data strobes /DS_N, indicating the physical transfer size requested (however, cachable slaves must always supply all 32 bits of data). The slave responds by driving data onto the bus, and then asserting /DTACK. The bus master then terminates the cycle by negating /FCS, at which point the slave will negate its

/SLAVEN line and tri-state its data. The cycle is done at this point. There are a few actions that modify a cycle termination, those will be covered in later sections.

The write cycle starts out the same way, up until DOE is asserted. At this point, it's the master that must drive data onto the bus, and then assert at least one /DSN line to indicate to the slave that data is valid and which data bytes are being written. The slave has the data for its use until it terminates the cycle by asserting /DTACK, at which point the master can negate /FCS and tri-state its data at any point. For maximum bus bandwidth, the slave can latch data on the falling edge of the logically ORed data strobes; the bus master doesn't sample /DTACK until after the data strobes are asserted, so a slave can actually assert /DTACK any time after /FCS.

3.2 Advanced Mode Support Logic

The Zorro III bus provides support for some more advanced things that weren't generally handled correctly on the Zorro II bus. Amiga computers have traditionally been supporting things that the more mainstream personal computers haven't. High speed DMA transfers and expansion coprocessors such as the Bridge Cards have been with the Amiga since the early days, and high performance main system CPUs with cache memory are now becoming common. The Zorro II bus never properly or easily supported such devices; the Zorro III bus attempts to make support of cache and coprocessor both possible and relatively straightforward. Other new features are covered in later sections.

3.2.1 Bus Locking

The first advanced modification of the basic bus cycle is bus locking, via the /LOCK signal. Bus locking is a hardware convention that allows a bus master to guarantee several cycles will be atomic on the bus. This is necessary to support the sharing of special "mail-box" memory between a bus master and an alternate PIC-based processor; Bridge Cards are an example of this kind of device. The Zorro II bus itself supports bus locking via the 68000 convention. However, the 68000 style of bus locking is often difficult to implement, and support for it was often ignored in Zorro II designs, especially those not directly concerned with multiprocessor support.

The Zorro III mechanism involves no change to the basic bus cycle, other than the monitoring of this /LOCK signal, and as such is much more reasonable to support. The /LOCK signal is asserted by a bus master at address time and maintained across cycles to lock out shared memory coprocessors, allowing hardware backed semaphores to easily be used between such coprocessors. We expect multiprocessing will be a greater concern on the Zorro III bus than it is at present; video coprocessors, RISC devices, and special purpose processors for image processing or mathematics should find a comfortable home on the Zorro III bus.

3.2.2 Cache Support

The other advanced cycle modifier on the Zorro III bus is the cache inhibit line, /CINH. On the Zorro II bus, there was originally no caching envisioned, and therefore no real support for

caching of Zorro II PICs. First in the A2630 and later in the Zorro III bus's emulation of Zorro II, conventions were adopted to permit caching of Zorro II cards. These conventions aren't perfect; MMU tables will sometimes have to supplant this geographic mapping. While Zorro III doesn't have any cache consistency mechanisms for managing caches between several caching bus masters, it does allow cards that absolutely must not be cached to assert a cache inhibit line, /CINH, on a per-cycle basis (asserted at slave time by a responding slave). This cache management is basically the lowest level of a cache management system, mainly useful for support of I/O and other devices that shouldn't be cached. Software will be required for the higher levels of cache management.

3.3 Multiple Transfer Cycles

The multiplexed address/data design of the Zorro III bus has some definite advantages. It allows Zorro III cards to use the same 100-pin connector as the Zorro II cards, which results in every bus slot being a 32-bit slot, even if there's an alternate connector in-line with any or all of the system slots; current alternate connectors include Amiga Video and PC-AT (now sometimes called ISA, for *Industry Standard Architecture*, now that it's basically beyond the control of IBM) compatible connectors. This design also makes implementation of the bus controller for a system such as the A3000 simpler. And it can result in lower cost for Zorro III PICs in many cases.

The main disadvantage of the multiplexed bus is that the multiplexing can waste time. The address access time is the same for multiplexed and non-multiplexed buses, but because of the multiplexing time, Zorro III PICs must wait until *data time* to assert data, which places a fixed limit on how soon data can be valid. The Zorro III Multiple Transfer Cycle is a special mode designed to allow the bus to approach the speed of a non-multiplexed design. This mode is especially effective for high speed transfers between memory and I/O cards.

As the name implies, the Multiple Transfer Cycle is an extension of the basic full cycle that results in multiple 32-bit transfers. It starts with a normal full cycle *address phase* transaction, where the bus master drives the 32-bit address and asserts the /FCS signal. A master capable of supporting a Multiple Transfer Cycle will also assert /MTCR at the same time as /FCS. The slave latches the address and responds by asserting its /SLAVEN line. If the slave is capable of multiple transfers, it'll also assert /MTACK, indicating to the bus master that it's capable of this extended cycle form. If either /MTCR or /MTACK is negated for a cycle, that cycle will be a basic full cycle.

Assuming the multiple transfer handshake goes through, the multiple cycle continues to look similar to the basic cycle into the data phase. The bus master asserts DOE (possibly with write data) and the appropriate /DSN, then the slave responds with /DTACK (possibly with read data at the same time), just as usual. Following this, however, the cycle's character changes. Instead of terminating the cycle by negating /FCS, /DSN, and DOE, the master negates /DSN and /MTCR, but maintains /FCS and DOE. The slave continues to assert /SLAVEN, and the bus goes into what's called a *short cycle*.

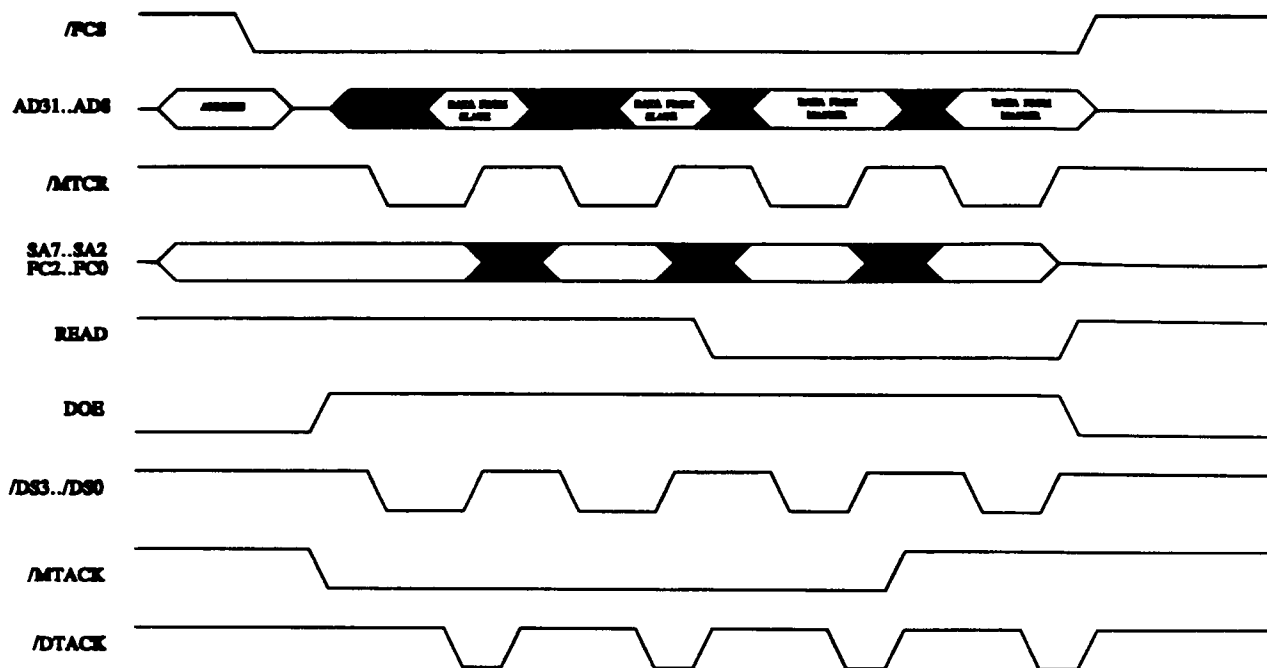


Figure 3-2: Multiple Transfer Cycles

The short cycle begins with the bus master driving the low order address lines A7-A2; these are the non-multiplexed addresses and can change without a new *address phase* being required (this is essentially a page mode, fully random accesses on this 256 byte page). The READ line may also change at this time. The master will then assert /MTCR to indicate to the slave that the short cycle is starting. For reads, the appropriate /DS_N are asserted simultaneously with /MTCR, for writes, data and /DS_N are asserted slightly after /MTCR. The slave will supply data for reads, then assert /DTACK, and the bus will terminate the short cycle and start into either another short cycle or a full cycle, depending on the multiple cycle handshaking that has taken place.

The question of whether a subsequent cycle will be a full cycle or a short cycle is answered by multiple cycle arbitration. If the master can't sustain another short cycle, it will negate /FCS and DOE along with /MTCR at the end of the current short cycle, terminating the full cycle as well. The master always samples the state of /MTACK on the falling edge of /MTCR. If a slave can't support additional short cycles, it negates /MTACK one short cycle ahead of time. On the following short cycle, the bus master will see that no more short cycles can be handled by the slave, and fully terminate the multiple transfer cycle once this last short cycle is done.

PICs aren't absolutely required to support Multiple Transfer Cycles, though it is a highly recommended feature, especially for memory boards. And of course, all PICs must act intelligently about such cycles on the bus; a card doesn't request or acknowledge any Multiple Transfer Cycle it can't support.

3.4 Quick Bus Arbitration

The Zorro II bus does an adequate job of supporting multiple bus masters, and the Zorro III bus extends this somewhat by introducing fair arbitration to Zorro II cards. However, some desirable features cannot be added directly to the Zorro II arbitration protocol. Specifically, Zorro III bus arbitration is much faster than the Zorro II style, it prohibits bus hogging that's possible under the Zorro II protocol, and it supports intelligent bus load balancing.

Load balancing requires a bit of explanation. A good analogy is to that of software multitasking; there, an operating system attempts to slice up CPU time between all tasks that need such time; here, a bus controller attempts to slice up bus time between all masters that need such time. With preemptive multitasking such as in the Amiga and UNIX OSs, equal CPU time can be granted to every task (possibly modified by priority levels), and such scheduling is completely under control of the OS; no task can hog the CPU time at the expense of all others. An alternate multitasking scheme is a popular add-on to some originally non-multitasking operating systems lately. In this scheme, each task has the CPU until it decides to give up the CPU, basically making the effectiveness of the CPU sharing at the mercy of each task. This is exactly the same situation with masters on the Zorro II bus. The Zorro III arbitration mechanism attempts to make bus scheduling under the control of the bus controller, with masters each being scheduled on a cycle-by-cycle basis.

When a Zorro III PIC wants to master the bus, it *registers* with the bus controller. This tells the bus controller to include that PIC in its scheduling of the expansion bus. There may be any number of other PICs registered with the bus controller at any given time. The CPU is always scheduled expansion bus time, and other local bus devices, such as a hard disk controller, may be registered from time to time.

Once registered, a PIC sits idle until it receives a *grant* from the bus controller. A grant is permission from the bus controller that allows the PIC to master the Zorro III bus for one full cycle. A PIC always gets one full cycle of bus time when given a grant, and assuming it stays registered, it may receive additional full cycles. Within the full cycle, the PIC may run any number of Multiple Transfer Cycles, assuming of course the responding slave supports such cycles. For multiprocessor support, a PIC will be granted multiple atomic full cycles if it locks the bus. This feature is *only* for support of hardware semaphores and other such multiprocessor needs; it is not intended as a means of bus hogging!

Figure 3-3 shows the basics of Zorro III bus arbitration, which is pretty simple. While it uses some of the same signals as the 680x0 inspired Zorro II bus arbitration mechanism, it has nothing to do with 680x0 bus arbitration; the /BRN and /BGN signals should be thought of as completely new signals. In order to register with the bus controller as a bus master, a PIC asserts its private /BRN strobe on the rising edge of the 7M clock, and negates it on the next rising edge. The bus controller will indicate mastership to a registered bus master by asserting its /BGN. Once granted the bus, the PIC drives only the standard cycle signals: addresses, /FCS, /EDSN, data, etc. in a full cycle. The bus controller manages the assertion of /OWN and /BGACK, which are important only for bus management and Zorro II support. While a scheduling scheme

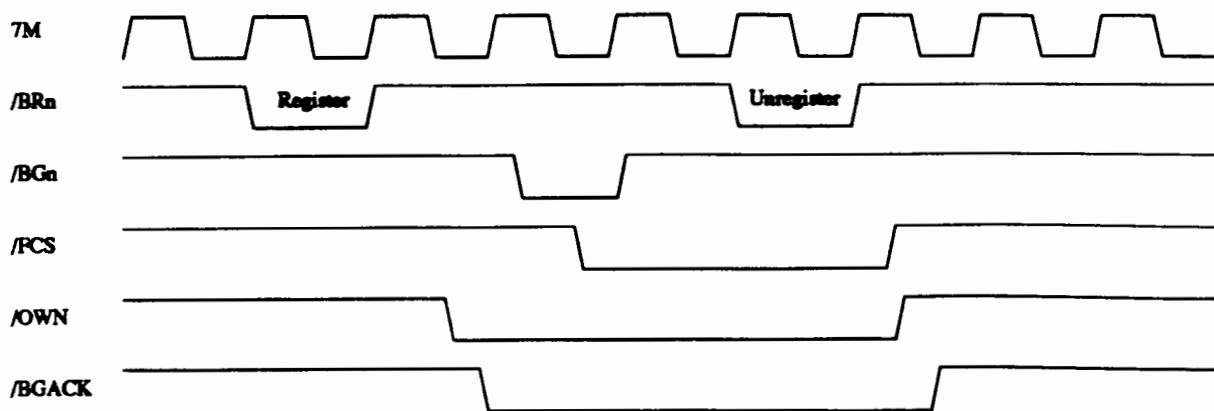


Figure 3-3: Zorro III Bus Arbitration

isn't part of this bus specification, the bus master will only be guaranteed one bus cycle at a time. The /BGn line is negated shortly after the master asserts /FCS unless the bus controller is planning to grant multiple full cycles to the master. The only thing that'll force the controller to grant multiple full cycles is a locked bus. Any master that works better with multiple cycles, such as devices with buffers to empty into memory, should run a Multiple Transfer Cycle to transfer several longwords during the same full cycle. For this reason, slave cards are encouraged to support Multiple Transfer Cycles, even if they don't necessarily run any faster during them.

Once a registered bus master has no more work to do, it unregisters with the bus controller. This works just like registering -- the PIC asserts /BRn on the rise of 7M, then negates it on next rising 7M. This is best done during the last cycle the bus master requires on the bus. If a registered master gets a grant before unregistering and has no work to do, it can unregister without asserting /FCS, to give back the bus without running a cycle. It's always far better to make sure that the master unregisters as quickly as possible. Bus timeout causes an automatic unregistering of the registered master that was granted that timed-out cycle; this guarantees that an inactive registered master can't drag down the system. If a master sees a /BERR during a cycle, it should terminate that cycle immediately and re-try the same cycle. If the retried cycle results in a /BERR as well, nothing more can be done in hardware; notification of the driver program is the usual recourse.

The bus controller may have to mix Zorro II style bus arbitration in with Zorro III arbitration, as Zorro II and Zorro III cards can be freely mixed in a backplane. Because of this, Multiple Transfer Cycles, and the self-timed nature of Zorro III cards, there's no way to guarantee the latency between bus grants for a Zorro III card. The bus controller does, however, make sure that all masters are fairly scheduled so that no starvation occurs, if at all possible. Zorro III cards must use Zorro III style bus arbitration; although current Zorro III backplanes can't differentiate between Zorro II and Zorro III cards when they request (other than by the request mechanism), it can't be assumed that a backplane will support Zorro III cycles with Zorro II mastering, or vice-versa.

3.5 Quick Interrupts

While the Zorro II bus has always supported shared interrupts, the Zorro III bus supports a mechanism wherein the interrupting PIC can supply its own vector. This has the potential to make such vectored interrupts much faster than conventional Zorro II chained interrupts, arbitrating the interrupting device in hardware instead of software.

A PIC supporting quick interrupts has on-board registers to store one or more vector numbers; the numbers are obtained from the OS by the device driver for the PIC, and the PIC/driver combination must be able to handle the situation in which no additional vectors are available. During system operation, this PIC will interrupt the system in the normal manner, by

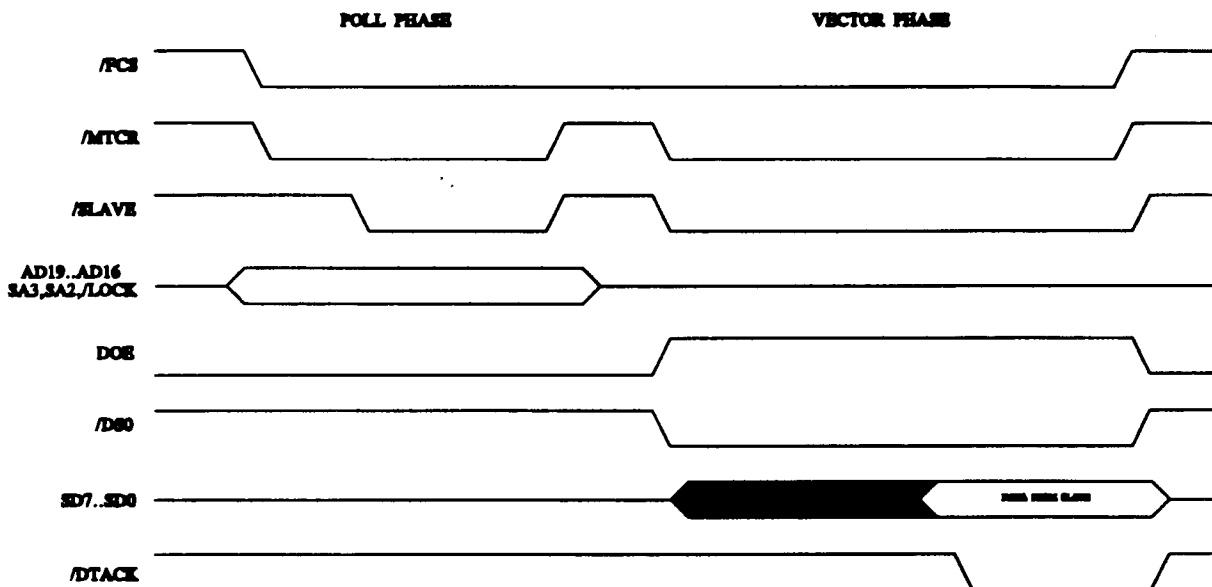


Figure 3-4: Interrupt Vector Cycle

asserting one of the bus interrupt lines. This interrupt will cause an interrupt vector cycle to take place on the bus. This cycle arbitrates in hardware between all PICs asserting that interrupt, and it's a completely different type of Zorro III cycle, as illustrated in *Figure 3-4*.

The bus controller will start an interrupt vector cycle in response to an interrupt asserted by any PIC. This cycle starts with /FCS and /MTCR asserted, a FC code of 7 (CPU space), a CPU space cycle type, given by address lines A16-A19, of 15, and the interrupt number, which is on A1-A3 (A1 is on the /LOCK line, as in Zorro II cycles). The interrupt numbers 2 and 6 are currently defined, corresponding to /INT₂ and /INT₆ respectively; all others are reserved for future use. At this point, called the *polling phase*, any PIC that has asserted an interrupt and wants to supply a vector will decode the FC lines, the cycle type, match its interrupt number against the one on the bus, and assert /SLAVEN if a match occurs. Shortly thereafter, the /MTCR line is negated, and the slaves all negate /SLAVEN. But the cycle doesn't end.

The next step is called the *vector phase*. The bus controller asserts one /SLAVEN back to one of the interrupting PICs, along with /MTCR and /DS0, but no addresses are supplied. That

PIC will then assert its 8-bit vector onto the logical Do-D7 (physically AD15-AD8) of the 32-bit data bus and /DTACK, as quickly as possible, thus terminating the cycle. The speed here is very critical; an automatic autovector timeout will occur very quickly, as any actual waiting that's required for the quick interrupt vector is potentially delaying the autovector response for Zorro II style interrupts. A PIC stops driving its interrupt when it gets the response cycle; it must also be possible for this interrupt to be cleared in software (e.g., the PIC must make choice of vectoring vs. autovectoring a software issue).

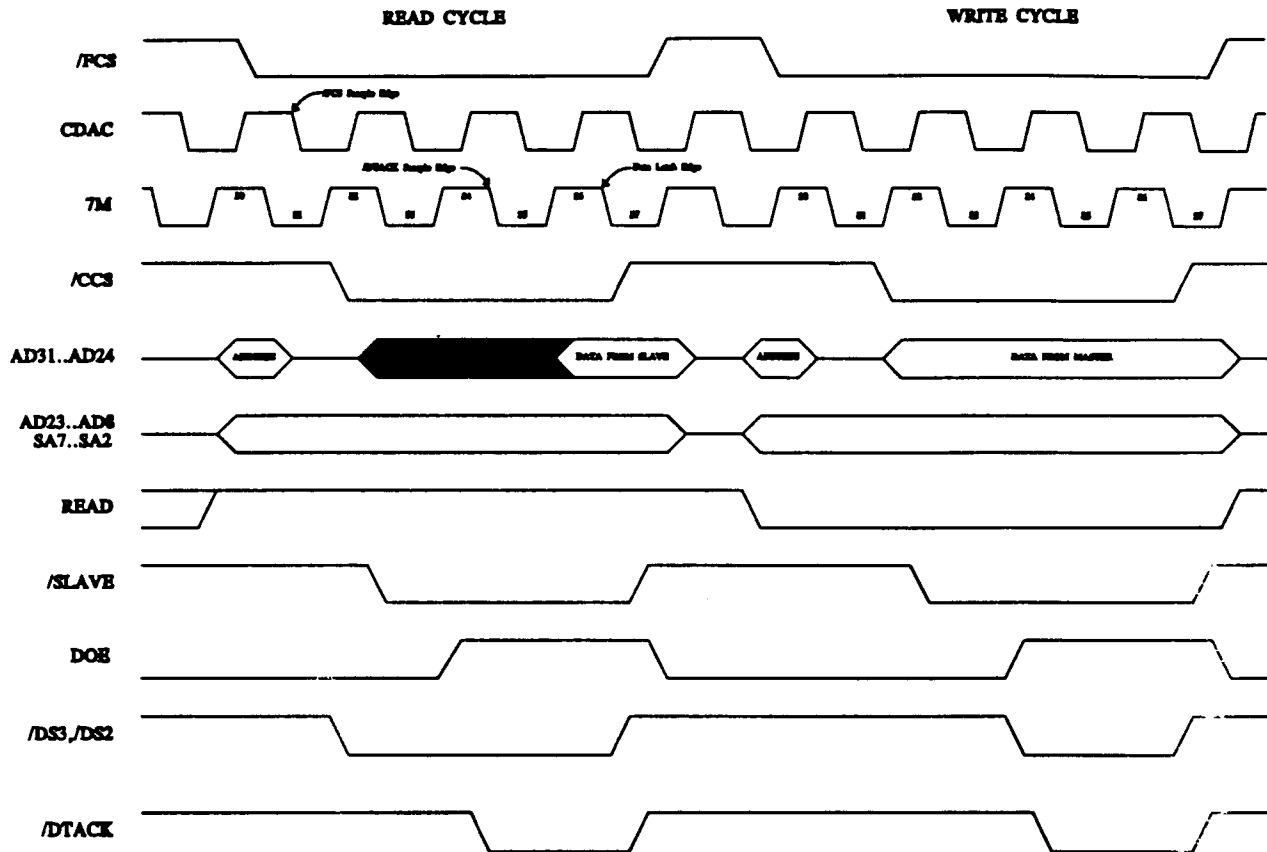


Figure 3-5: Zorro II Within Zorro III

3.6 Compatibility with Zorro II Devices

As detailed in Chapter 2, the Zorro III bus supports a bus cycle mode very similar to the 68000-based Zorro II bus, and is expected to be compatible with all properly designed Zorro II PICs. As shown in *Figure 1-1*, Zorro II and Zorro III expansion spaces are geographically mapped on the Zorro III bus. The mapping logic resides on the bus, and operates on the bus address presented for any cycle. Every cycle starts out assuming a Zorro III cycle, but the mapping logic will inscribe a Zorro II cycle within the Zorro III cycle if the address range is right. *Figure 3-5* details the bus action for this mode.

The cycle starts out with the usual address phase activity; the bus master asserts /FCS after asserting the full 32-bit address onto the address bus. The bus decoder maps the bus

address asynchronously and quickly, so that by the time /FCS is asserted, the memory space is determined. A Zorro II space access will cause A8-A23 to remain asserted, rather than being tri-stated along with A24-A31, as the Zorro III cycle normally does. The bus controller synchs the asynchronous /FCS on the falling edge of CDAC, then drives /CCS (the /AS equivalent) out on the rising edge of 7M, based on that synched /FCS. For a read cycle, /DS3 and/or /DS2 (the /UDS and /LDS replacements, respectively) would be asserted along with /CCS; write cycles see those lines asserted on the next rising edge of 7M, at S4 time. The DOE line is also asserted at the start of S4.

The bus controller starts to sample /DTACK on the falling edge of 7M between S4 and S5, adding wait states until /DTACK is encountered. As per Zorro II specs, the PIC need not create a /DTACK unless it needs that level of control; there are Zorro II signals to delay the controller-generated /DTACK, or take it over when necessary. The controller will drive its automatic /DTACK at the start of S4, leaving plenty of time for the sampling to come at S5. Once a /DTACK is encountered, cycle termination begins. The controller latches data on the falling 7M edge between S6 and S7, and also negates /CCS and the /DSn at this time. Shortly thereafter, the controller negates /DTACK (when controlling it), DOE, and tri-states the data bus, getting ready for the next cycle.

3.7 Zorro III Implementations

Functionally, there are two possible implementation levels in existence for the Zorro III bus. All of the features described in this document are required for a full compliance Zorro III bus. However, the original Amiga 3000 computers were shipped with a bus controller that supported only a subset of the Zorro III specification published here. This is, however, upgradable.

The A3000 implementation of the Zorro III bus is driven by a custom controller chip called **Fat Buster**. The specification of this chip and the A3000 hardware are fully capable of supporting the complete Zorro III bus, but the initial silicon on Fat Buster, called the Level 1 Fat Buster, omits some features. Missing are:

- Support of Multiple Transfer Cycles.
- Support for Zorro III style bus arbitration.
- Support for Quick Interrupts.

The Level 2 version of Fat Buster has been in testing for some time at Commodore in West Chester, PA. Any developers who immediately intend to design PICs supporting these features are urged to contact Commodore Amiga Technical Support/Amiga Developer Support Europe for more information on obtaining samples of this part for use in A3000 systems. These parts are likely to be introduced into production, and available as part of an A3000 upgrade, very soon. All Buster chip revisions "13G" and earlier support the Level 1 features. Buster chip revisions "13H" and later support Level 2 features and improved Level 1 features as well.

—

—

—

CHAPTER 4

SIGNAL DESCRIPTION

*"Pushing back the limits of human achievement, reaching for the stars,
that's not something we do. It's what we are."*

-Michael Swaine

The signals detailed here are the Zorro III mode signals. While some of this information is the same in as the Zorro II signal description of Chapter 2, many like-seeming bus signals behave differently in Zorro III mode than Zorro II mode. These can be a very important differences; thus the complete set of signals is detailed here.

4.1 Power Connections

The expansion bus provides several different voltages designed to supply expansion devices. These are basically the same for the Zorro III bus as they were for the Zorro II bus, with the exception of one pin, and that the specification has been clarified a bit. Note that all Zorro III PICs must list their power consumption specifications.

Digital Ground (Ground)

This is the digital supply ground used by all expansion cards as the return path for all expansion supplies.

Main Supply (+5VDC)

This is the main power supply for all expansion cards, and it is capable of sourcing large currents; each PIC can draw up to 2.0 Amps @ +5VDC.

Negative Supply (-5VDC)

This is a negative version of the main supply, for small current loads only; each PIC can draw up to 60 mA @ -5VDC.

High Voltage Supply (+12VDC)

This is a higher voltage supply, useful for communications cards and other devices requiring greater than digital voltage levels. This is intended for relatively small current loads only; each PIC can draw up to 500mA @ +12VDC.

Negative High Supply (-12VDC)

Negative version of the high voltage supply, also used in communications applications, and similarly intended for small loads only; each PIC can draw up to 60 mA @ -12VDC.

4.2 Clock Signals

The expansion bus provides clock signals for expansion boards. The main use for these clocks on Zorro III cards is bus arbitration clocking. There is no relationship between any of these clocks and normal Zorro III bus activity. The relationship between these clocks is illustrated in *Figure 2-2*.

/C1 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the falling edge of the 7M system clock.

/C3 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the rising edge of the 7M system clock.

CDAC Clock

This is a 7.16 MHz system clock (7.09 MHz on PAL systems) which trails the 7M clock by 90° (approximately 35ns).

E Clock

This is the 68000 generated "E" clock, used for 6800 family peripherals driven by "E" and 6502 peripherals driven by Φ_2 . This clock is four 7M clocks high, six clocks low, as per the 68000 spec.

7M Clock

This is the 7.16 MHz system clock (7.09 MHz on PAL systems). This clock drives the bus master registration mechanism for Zorro III bus masters.

4.3 System Control Signals

The signals in this group are available for various types of system control; most of these have an immediate or near immediate effect on expansion cards and/or the system CPU itself.

Hardware Bus Error/Interrupt (/BERR)

This is a general indicator of a bus fault or special condition of some kind. Any expansion card capable of detecting a hardware error relating directly to that card can assert /BERR when that bus error condition is detected, especially any sort of harmful hardware error condition. This signal is the strongest possible indicator of a bad situation, as it causes all PICs to get off the bus, and will usually generate a level 2 exception on the host CPU. For any condition that can be handled in software and doesn't pose an immediate threat to hardware, notification via a standard processor interrupt is the better choice. The bus controller will drive /BERR in the event of a detected bus collision or DMA error (an attempt by a bus master to access local bus resources it doesn't have valid access permission for). All cards must monitor /BERR and be prepared to tri-state all of their on-bus output buffers whenever this signal is asserted. An expansion bus master will attempt to retry a cycle aborted by a single /BERR and notify system software in the case of two subsequent /BERR results. Since any number of devices may assert /BERR, and all bus cards must monitor it, any device that drives /BERR must drive with an open collector or similar device, and any device that monitors /BERR should place a minimal load on it. This signal is pulled high by a passive backplane resistor.

Note that, especially for the slave device being addressed, that /BERR alone is not always necessarily an indication of a bus failure in the pure sense, but may indicate some other kind of unusual condition. Therefore, a device should still respond to the bus address, if otherwise appropriate, when a /BERR condition is indicated. It simply tri-states its bus buffers and other outputs, and waits for a change in the bus state. If the /BERR signal is negated with the cycle unterminated, the special condition has been resolved and the slave responds to the rest of the cycle as it normally would have. If the cycle is terminated by the bus master, the resolution of the special condition has indicated that the addressed slave is not needed, and so the cycle terminates without the slave being used.

System Reset (/RESET, /IORST)

The bus supplies two versions of the system reset signal. The /RESET signal is bidirectional and unbuffered, allowing an expansion card to hard reset the system. It should only be used by boards that need this reset capability, and is driven only by an open collector or similar device. The /IORST signal is a buffered output-only version of the reset signal that should be used as the normal reset input to boards not concerned with resetting the system on their own. All expansion devices are required to reset their autoconfiguration logic when /IORST is asserted. These signals are pulled high by passive backplane resistors.

System Halt (/HLT)

This signal is driven, along with /RESET, to assert a full-system reset. A full-system reset is asserted on a powerup reset or a keyboard reset; any PIC that needs to differentiate between full system and I/O reset should monitor /HLT and /IORST unless it also needs to drive a reset condition. This is driven with an open-collector output, or the equivalent, and pulled up by a backplane resistor.

System Interrupts

Two of the decoded, level sensitive 680x0 interrupt inputs are available on the expansion bus, and these are labelled as /INT₂ and /INT₆. Each of these interrupt lines is shared by wired ORing, thus each line must be driven by an open-collector or equivalent output type. Zorro III interrupts can be handled Zorro II style, via autovectors and daisy-chained polling, or they can be vectored using the quick interrupt protocol described in Chapter 3. Zorro II and Zorro III systems originally provided /INT₁, /INT₄, /INT₅, and /INT₇ lines as well, but as these were never properly supportable by system software, they have been eliminated, those lines now considered reserved for future use in a Zorro III system.

4.4 Slot Control Signals

This group of signals is responsible for the control of things that happen between expansion slots.

Slave (/SLAVEN)

Each slot has its own /SLAVEN output, driven actively, all of which go into the collision detect circuitry. The "N" refers to the expansion slot number of the particular /SLAVE signal. Whenever a Zorro III PIC is responding to an address on the bus, it must assert its /SLAVEN output very quickly. If more than one /SLAVEN output occurs for the same address, or if a PIC asserts its /SLAVEN output for an address reserved by the local bus, a collision is registered and the bus controller asserts /BERR. The bus controller will assert /SLAVEN back to the interrupting device selected during a Quick Interrupt cycle, so any device supporting Quick Interrupts must be capable of tri-stating its /SLAVEN; all others can drive SLAVEN with a normal active output.

Configuration Chain (/CFGINN, /CFGOUTN)

The slot configuration mechanism uses the bus signals /CFGOUTN and /CFGINN, where "N" refers to the slot number. Each slot has its own version of both signals, which make up the *configuration chain* between slots. Each subsequent /CFGINN is a result of all previous /CFGOUTs, going from slot 0 to the last slot on the expansion bus. During the autoconfiguration process, an unconfigured Zorro III PIC responds to the 64K address space starting at either \$00E80000 or \$FF000000 if its /CFGINN signal is asserted. All unconfigured PICs start up with /CFGOUTN negated. When configured, or told to "shut up", a PIC will assert its /CFGOUTN, which results in the /CFGINN of the next slot being asserted. Backplane logic automatically passes on the state of the previous /CFGOUTN to the next /CFGINN for any slot not occupied by a PIC, so there's no need to sequentially populate the expansion bus slots.

Backplane Type Sense (SenseZ3)

This line can be used by the PIC to determine the backplane type. It is grounded on a Zorro II backplane, but floating on a Zorro III backplane. The Zorro III PIC connects this signal to a 1K pullup resistor to generate a real logic level for this line. It's possible, though more complicated, to build a Zorro III PIC that can actually run in Zorro II mode when in a Zorro II backplane. It's hardly necessary or required to support this backward

compatibility mechanism, and in many cases it'll be impractical. The Zorro III specification does require that this signal be used, at least, to shut the card down and pass /CFGIN to /CFGOUT when in a Zorro II backplane.

4.5 DMA Control Signals

There are various signals on the expansion bus that coordinate the arbitration of bus masters. Zorro II bus masters use some of the same logical signals, but their arbitration protocol is considerably different.

PIC is DMA Owner (/OWN)

This is asserted by the bus controller when a master is about to go on the bus and indicates that some master owns the bus. Zorro II bus masters drive this, and some Zorro III slaves may find a need to monitor it, or /BGACK, to determine who's the bus master. This is ordinarily not important to Zorro III PICs, and they may not drive this line.

Slot Specific Bus Arbitration (/BRN, /BGN)

These are the slot-specific /BRN and /BGN signals, where "N" refers to the expansion slot number. The bus request from each board is taken in by the bus controller and ultimately used to take over the system from the primary bus master, which is always the local master. Zorro III PICs toggle /BRN to register or unregister as a master with the bus controller. /BGN is asserted to one registered PIC at a time, on a cycle by cycle basis, to indicate to the PIC that it gets the bus for one full cycle.

Bus Grant Acknowledge (/BGACK)

Asserted by the bus controller when a master is about to go on the bus. As with /OWN, most Zorro III PICs ignore this signal, and none may drive it.

Bus Want/Clear (/BCLR)

This signal is asserted by the bus controller to indicate that a PIC wants to master the bus; Zorro III cards can use this to determine if any Zorro II bus requests are pending; Zorro III bus requests don't affect /BCLR.

4.6 Address and Related Control Signals

These signals are various items used for the addressing of devices in Zorro III mode by bus masters either on the bus or from the local bus. The bus controller translates local bus signals (68030 protocol on the A3000 and A4000) into Zorro III signals; masters are responsible for creating the appropriate signals via their own bus control logic.

Read Enable (READ)

Read enable for the bus; READ is asserted by the bus master during a bus cycle to indicate a read cycle, READ is negated to indicate a write cycle. READ is asserted at address time, prior to /FCS, for a full cycle, and prior to /MTCR for a short cycle. READ stays valid throughout the cycle; no latching required.

Multiplexed Address Bus (A8-A31)

These signals are driven by the bus master during address time, prior to the assertion of /FCS. Any responding slave must latch as many of these lines as it needs on the falling edge of /FCS, as they're tri-stated very shortly after /FCS goes low. These addresses always include all configuration address bits for normal cycles, and the cycle type information for Quick Interrupt cycles.

Short Address Bus (A2-A7)

These signals are driven by the bus master during address time, prior to the assertion of /FCS, for full cycles, and prior to the assertion of /MTCR for short cycles. They stay valid for the entire full or short cycle, and as such do not need to be latched by responding slaves.

Table 4-1: Memory Space Type Codes

FC ₀	FC ₁	FC ₂	Address Space Type	Z3 Response
0	0	0	Reserved	None
0	0	1	User Data Space	Memory
0	1	0	User Program Space	Memory
0	1	1	Reserved	None
1	0	0	Reserved	None
1	0	1	Supervisor Data Space	Memory
1	1	0	Supervisor Program Space	Memory
1	1	1	CPU Space	Interrupts

Memory Space (FC₀-FC₂)

The memory space bits are an extension to the bus address, indicating which type of access is taking place. Zorro III PICs must pay close attention to valid memory space types, as the space type can change the type of the cycle driven by the current bus master. The encoding is the same as the valid Motorola function codes for normal accesses. These are driven at address time, and like the low short address, are valid for an entire short or full cycle.

Compatibility Cycle Strobe (/CCS)

This is equivalent to the Zorro II address strobe, /AS. A Zorro III PIC doesn't use this for normal operation, but may use it during the autoconfiguration process if configuring at the Zorro II address. AUTOCONFIG[®] cycles at \$00E8xxxx always look like Zorro II cycles, though of course /FCS and the full Zorro III address is available, so a card can use either Zorro II or Zorro III addressing to start the cycle. However, using the /CCS strobe can save the designer the need to compare the upper 8 bits of address. Data must be driven Zorro II style, though if the /DS_N lines are respected for reads, /CINH is asserted, and /MTACK is negated, the resulting Zorro III cycle will fit within the expected Zorro II cycle generated by the bus controller.

Yes, that should sound weird; it's based on the mapping of Zorro II vs. Zorro III signals, and of course the fact that $\overline{\text{FCS}}$ always starts any cycle. Also note that a bus cycle with $\overline{\text{CCS}}$ asserted and $\overline{\text{FCS}}$ negated is always a Zorro II PIC-as-master cycle. Many Zorro III cards will instead configure at the alternate $\$ \text{FF00xxxx}$ base address, fully in Zorro III mode, and thus completely ignore this signal.

Full Cycle Strobe ($\overline{\text{FCS}}$)

This is the standard Zorro III full cycle strobe. This is asserted by the bus master shortly after addresses are valid on the bus, and signals the start of any kind of Zorro III bus cycle. Shortly after this line is asserted, all the multiplexed addresses will go invalid, so in general, all slaves latch the bus address on the falling edge of $\overline{\text{FCS}}$. Also, $\overline{\text{BGN}}$ line is negated for a Zorro III mastered cycle shortly after $\overline{\text{FCS}}$ is asserted by the master.

4.7 Data and Related Control Signals

The data time signals here manage the actual transfer of data between master and slave for both full and short cycle types. The burst mode signals are here too, as they're basically data phase signals even though they don't only concern the transfer of data.

Data Output Enable ($\overline{\text{DOE}}$)

This signal is used by an expansion card to enable the buffers on the data bus. The bus master drives this line to keep slave PICs from driving data on the bus until *data time*.

Data Bus (D0-D31)

This is the Zorro III data bus, which is driven by either the master or the slave when $\overline{\text{DOE}}$ is asserted by the master (based on READ). It's valid for reads when $\overline{\text{DTACK}}$ is asserted by the slave; on writes when at least one of $\overline{\text{DSN}}$ is asserted by the master, for all cycle types.

Data Strobes ($\overline{\text{DSN}}$)

These strobes fall during *data time*; $\overline{\text{DS3}}$ strobes D24-D31 , while $\overline{\text{DS0}}$ strobes D0-D7 . For write cycles, these lines signal data valid on the bus. At all times, they indicate which bytes in the 32 bit data word the bus master is actually interested in. For cachable reads, all four bytes must be returned, regardless of the value of the sizing strobes. For writes, only those bytes corresponding to asserted $\overline{\text{DSN}}$ are written. Only contiguous byte cycles are supported; e.g. $\overline{\text{DS3-0}} = 2, 4, 5, 6, \text{ or } 10$ is invalid.

Data Transfer Acknowledge ($\overline{\text{DTACK}}$)

This signal is used to normally terminate a Zorro III cycle. The slave is always responsible for driving this signal. For a read cycle, it asserts $\overline{\text{DTACK}}$ as soon as it has driven valid data onto the data bus. For a write cycle, it asserts $\overline{\text{DTACK}}$ as soon as it's done with the data. Latching the data on writes may be a good idea; that can allow a slave to end the cycle before it has actually finished writing the data to its local memory.

Cache Inhibit (/CINH)

This line is asserted at the same time as /SLAVEN to indicate to the bus master that the cycle must not be cached. If a device doesn't support caching, it must assert /CINH and actually obey the /DSN byte strobes for read cycles. Conversely, if the device supports caching, /CINH is negated and the device returns all four bytes valid on reads, regardless of the actual supplied /DSN strobes.

Multiple Cycle Transfers (/MTCR/MTACK)

These lines comprise the Multiple Transfer Cycle handshake signals. The bus master asserts /MTCR at the start of *data time* if it's capable of supporting Multiple Transfer Cycles, and the slave asserts /MTACK with /SLAVEN if it's capable of supporting Multiple Transfer Cycles. If the handshake goes through, /MTCR strobes in the short address and write data as long as the full cycle continues.

CHAPTER 5

TIMING

*"When dealing with the insane, the best method is
to pretend to be sane."*

-Hermann Hesse

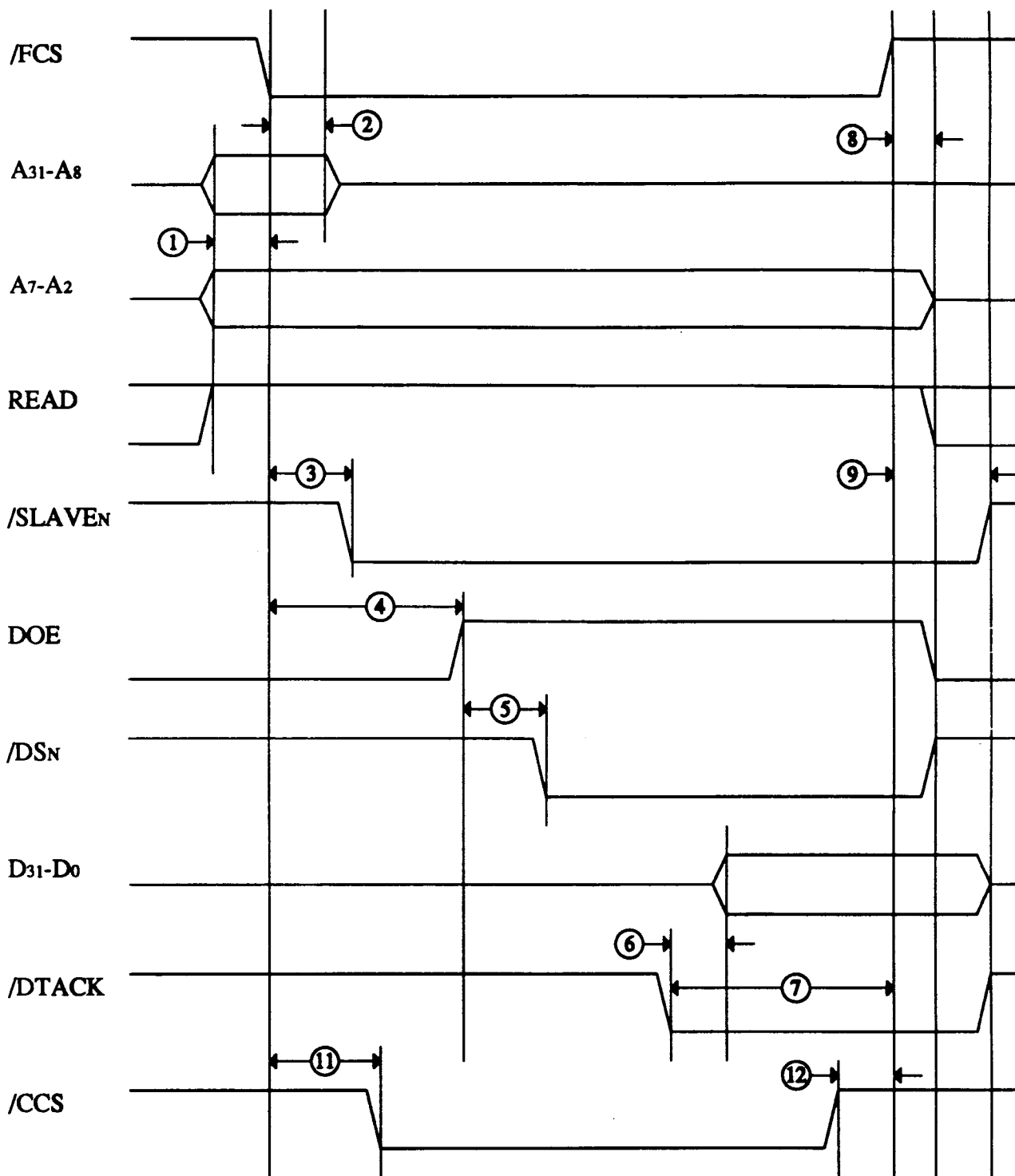
Some of this information is considered preliminary. Nothing is expected to get any more speed critical, but as mentioned previously, the testing of Zorro III designs has just started at the time of this writing, final bus controllers are not yet available, and only a few PIC designs have even been conceived.

This section covers the various timing specifications in detail for different Zorro III operations. It's important to realize that this timing information is a **specification**. Actual Zorro III systems may offer much more relaxed timings. Today. The whole point of the specification is that as long as all Zorro III PICs and all Zorro III backplanes base things on the timings given here, they'll always work together nicely. Any design based on the actual characteristics of any particular backplane will very likely wind up working only on that particular backplane.

The philosophy of timing on the Zorro III bus is to keep things as simple as possible without compromising the performance goals of the bus. Zorro III PICs are expected to be based on F-Series or ACT-series TTL logic, fast PALs, and possibly full custom chip designs. It's very unlikely the designer will meet any of these specifications with the LS parts left over from old Zorro II card designs.

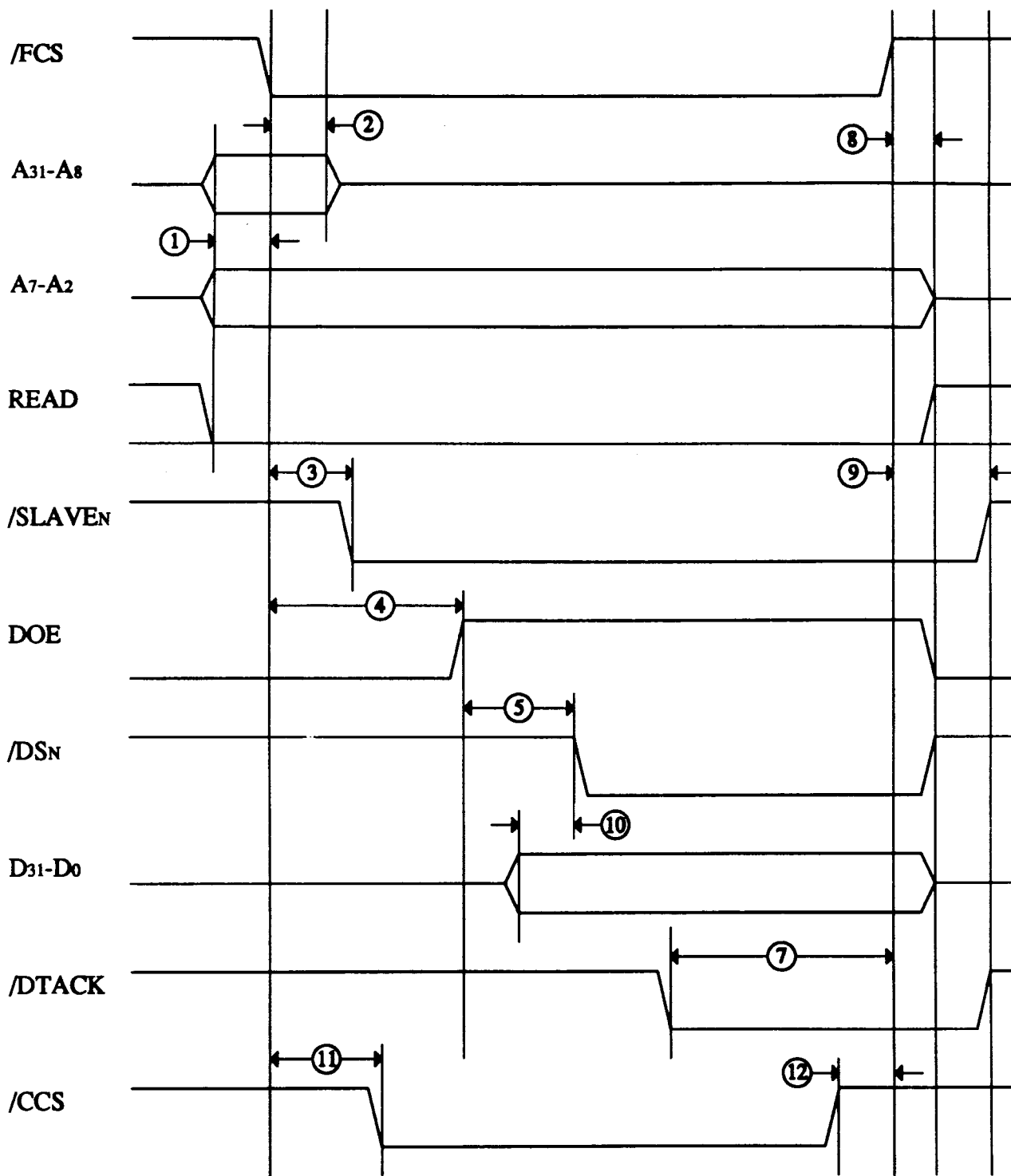
5.1 Standard Read Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVE _N delay	T _{SLV}	-----	25ns
4	/FCS to DOE delay	T _{DOE}	30ns	-----
5	DOE to /DS _N delay	T _{DS}	10ns	-----
6	Data setup to /DTACK	T _{RDS}	0ns	-----
7	/DTACK to /FCS off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
11	/FCS to /CCS delay	T _{CCS}	35ns	175ns
12	/CCS off to /FCS off	T _{ovL}	40ns	-----



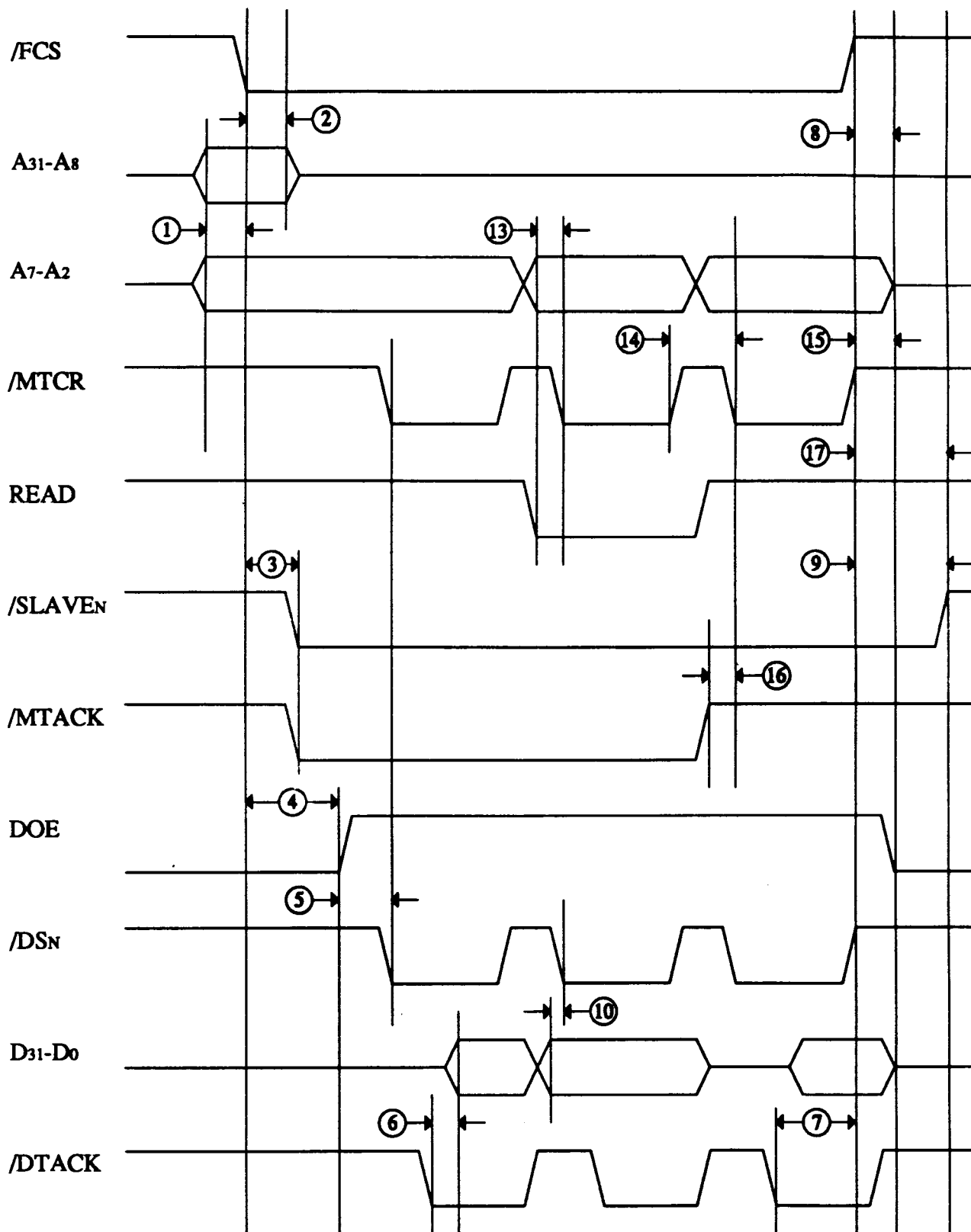
5.2 Standard Write Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVE _N delay	T _{SLV}	-----	25ns
4	/FCS to DOE delay	T _{DOE}	30ns	-----
5	DOE to /DS _N delay	T _{DS}	10ns	-----
7	/DTACK to /FCS off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
10	Write data setup to /DS _N	T _{WDS}	5ns	-----
11	/FCS to /CCS delay	T _{CCS}	35ns	175ns
12	/CCS off to /FCS off	T _{ovL}	40ns	-----



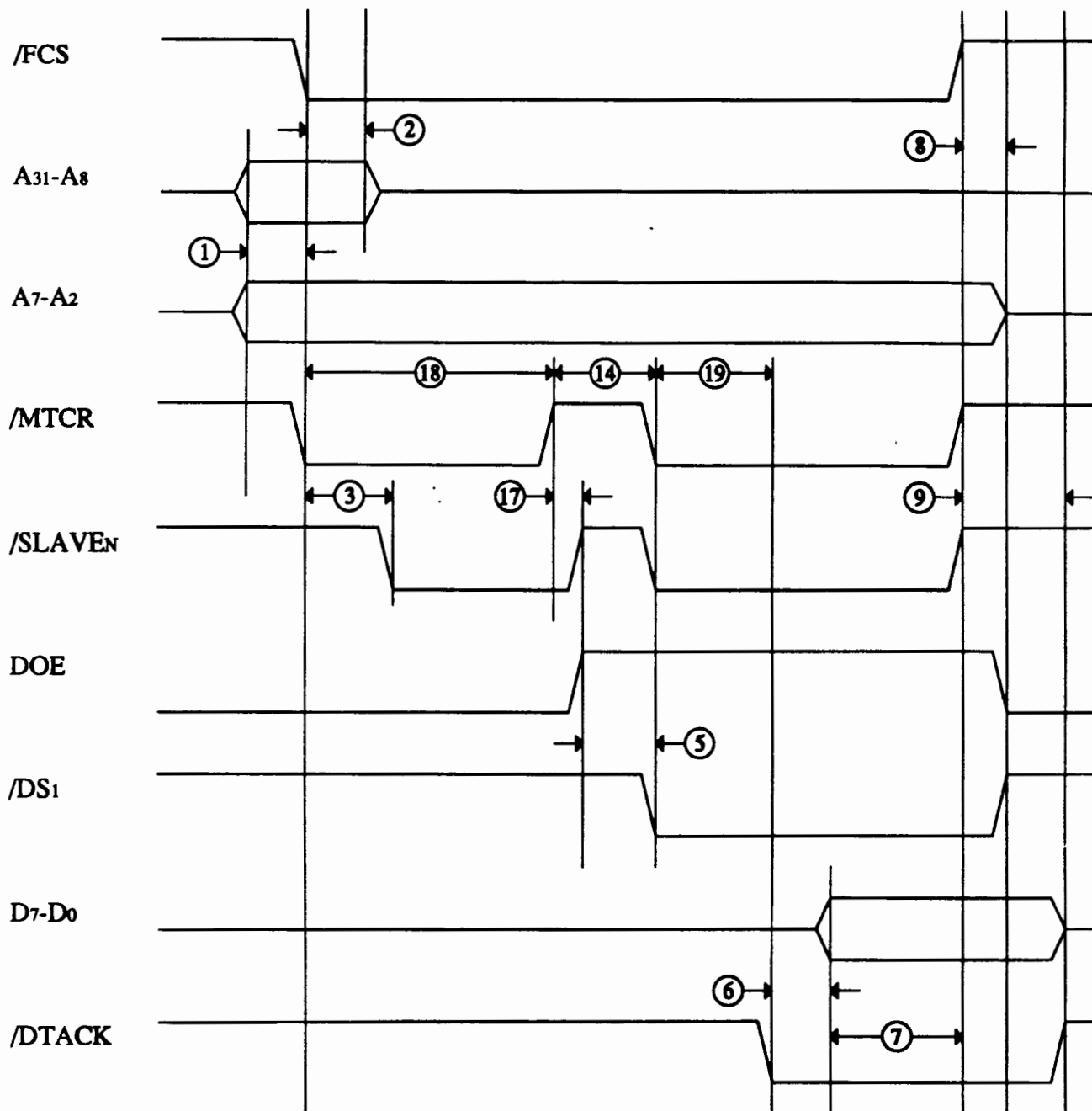
5.3 Multiple Transfer Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	----
2	Address hold from /FCS	T _{HAF}	10ns	----
3	/FCS to /SLAVE _N , /MTACK delay	T _{SLV}	----	25ns
4	/FCS to DOE delay	T _{DOE}	30ns	----
5	DOE to /DS _N , /MTCR delay	T _{DS}	10ns	----
6	Data setup to /DTACK	T _{RDS}	0ns	----
7	/DTACK to /FCS, /MTCR off	T _{OFF}	10ns	----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
10	Write data setup to /DS _N	T _{WDS}	5ns	----
13	Address, READ setup to /MTCR	T _{AMS}	5ns	----
14	/MTCR off to /MTCR on	T _{REF}	10ns	----
15	Address, READ hold from /MTCR	T _{HAM}	0ns	----
16	/MTACK off to /MTCR	T _{BCD}	10ns	----
17	Slave signal hold from /MTCR off	T _{HSM}	0ns	5ns



5.4 Quick Interrupt Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVEN delay	T _{SLV}	-----	25ns
5	DOE to /DSN delay	T _{DS}	10ns	-----
6	Data setup to /DTACK	T _{RDS}	0ns	-----
7	/DTACK to /FCS off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
14	/MTCR off to /MTCR on	T _{REF}	10ns	-----
17	Slave signal hold from /MTCR off	T _{HSM}	0ns	5ns
18	Poll Phase time	T _{POL}	30ns	100ns
19	Vector Phase start to /DTACK time	T _{VEC}	-----	100ns



CHAPTER 6

ELECTRICAL SPECIFICATIONS

"...I collected the instruments of life around me, that I might infuse a spark of being into the lifeless thing that lay at my feet"

-Victor Frankenstein

The Zorro III bus has a number of electrical specifications that are very important for PIC designers to consider, along with the timing parameters of course. It's extremely important to base designs on the specification of the backplane, rather than the actual behavior of the backplane. New backplanes for new machines are designed to conform to the specification, they are not necessarily based on previous designs. This is especially important with the Zorro III bus, since timing is far more critical than in the past, and the bus controller is designed from this specification, rather than the reverse, as in the Amiga 2000.

6.1 Expansion Bus Loading

The Zorro III bus loading is specified based on typical TTL family "F" series buffer devices, though in reality, compatible CMOS devices are likely to be used in some bus controllers or PICs. Thus, it's important to accept the TTL levels as a minimum voltage level, and make sure that all inputs are the appropriate TTL levels, while outputs can be at TTL or CMOS voltage levels as long as they provide the required source and sink.

While some A2000 designs used "LS" or "ALS" buffers instead of "F", the bus will generally work with these older cards, at least with current backplane designs such as the A3000 backplane. However, Zorro III designs must exactly obey these loading rules; it's very probable that some future Zorro III machines will have a large number of slots. In such machines, PICs built on the Zorro II specification will still work in a lightly loaded bus, but may not function in a fully loaded bus. All Zorro III PICs built to spec will work in any Zorro III backplane, without any loading problems, if all loading and timing rules are followed by the PIC designer. The bus

Table 6-1: Zorro III Drive Types

Signal	Direction	High Level	Low Level
Standard	Loading Driven	+140 μ A @ +2.7VDC +2.5VDC @ -3.0mA	-3.2mA @ +0.4VDC +0.4VDC @ +64mA
Clock	Loading	+20 μ A @ +2.7VDC	-1.6mA @ +0.4VDC
O.C.	Loading Driven	+80 μ A @ +2.7VDC Not Driven	-3.2mA @ +0.4VDC +0.4VDC @ +20mA
Non-bussed	Loading Driven	+80 μ A @ +2.7VDC +2.5VDC @ -0.4mA	-1.0mA @ +0.4VDC +0.4VDC @ +4.0mA

signals are divided up into the four groups shown in Table 6-1, based on the loading characteristics of the particular signal. The signals in each group are given here.

6.1.1 Standard Signals

The majority of signals on the bus are in this group. These are bussed signals, driven actively on the bus by F-series (or compatible) drivers such as 74F245, usually tri-stated when ownership of the signal changed for master and slave, and generally terminated with a 220 Ω /330 Ω thevenin terminator. PICs can apply two standard loads to each of these signals when necessary.

/FCS	/CCS	/DS0-/DS3	/LOCK
A2-A7	AD8-AD31	SD0-SD7	READ
FC0-FC2	DOE	/IORST	/BCLR
/MTCR	/MTACK		

6.1.2 Clock Signals

All clock signals on the bus are in this group. Many designs are very sensitive to clock delay, skew, and rise/fall times, so loading on the clock lines must be kept to a minimum. These are bussed signals, actively driven by the backplane, and source terminated with a low value

series resistor. PICs can apply one standard load to each of these signals when necessary. Zorro II cards have the same clock rules, so there should never be clocking problems when using either card type in a backplane.

/C3	CDAC	/C1	7M
E Clock			

6.1.3 Open Collector Signals

Many of the bus signals are shared via open collector or open drain outputs rather than via tri-stated signals; this is of course required for some asynchronous things like the shared interrupt lines, and it works well for other types of signals as well. Of course, a backplane resistor pulls these lines high, PICs only drive the line low.

/OWN	/BGACK	/CINH	/BERR
/DTACK	/RESET	/INT ₂	/INT ₆
/HLT			

6.1.4 Non-bussed Signals

The non-bussed, or slot specific, signals are involved with only one slot on the bus (eg, each slot has its own copy). As a result, the drive requirements are much less for these signals. The backplane provides pullups or pulldowns, as required by the specific signal.

/CFGIN _N	/CFGOUT _N	/BR _N	/BG _N
SenseZ ₃	/SLAVEN		

6.2 Slot Power Availability

The system power for the Zorro III bus is totally based on the slot configurations. A backplane is always free to supply extra power, but it must meet the minimum requirements specified here. All PICs must be designed with the minimum specifications in mind, especially the tolerances.

Pin	Supply
5,6	+5 VDC \pm 5% @ 2 Amps
8	-5 VDC \pm 5% @ 60 mA
10	+12 VDC \pm 5% @ 500mA
20	-12 VDC \pm 5% @ 60mA

6.3 Temperature Range

The Zorro III bus is specified for operation over a temperature range of 0° C to 70° C.

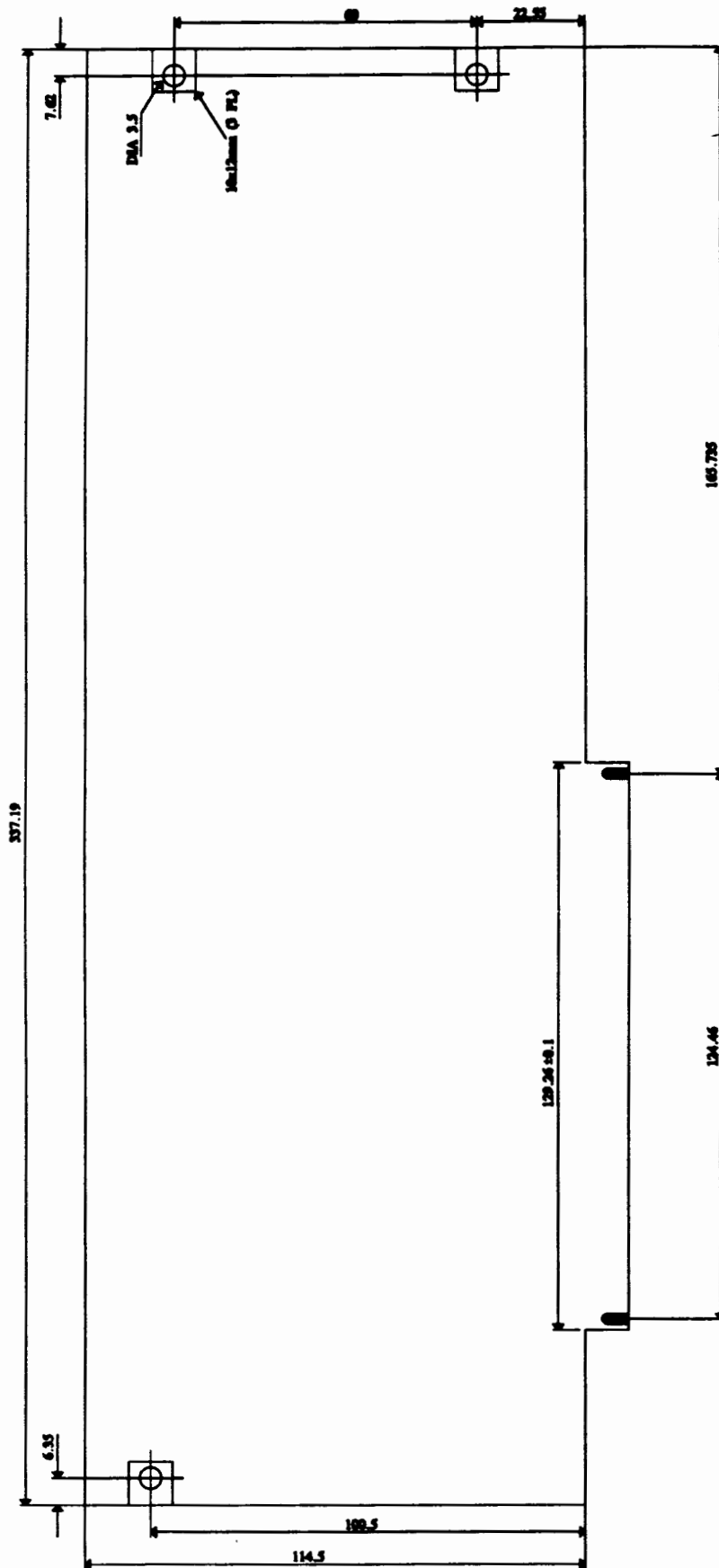
CHAPTER 7

MECHANICAL SPECIFICATIONS

"Never speak more clearly than you think."

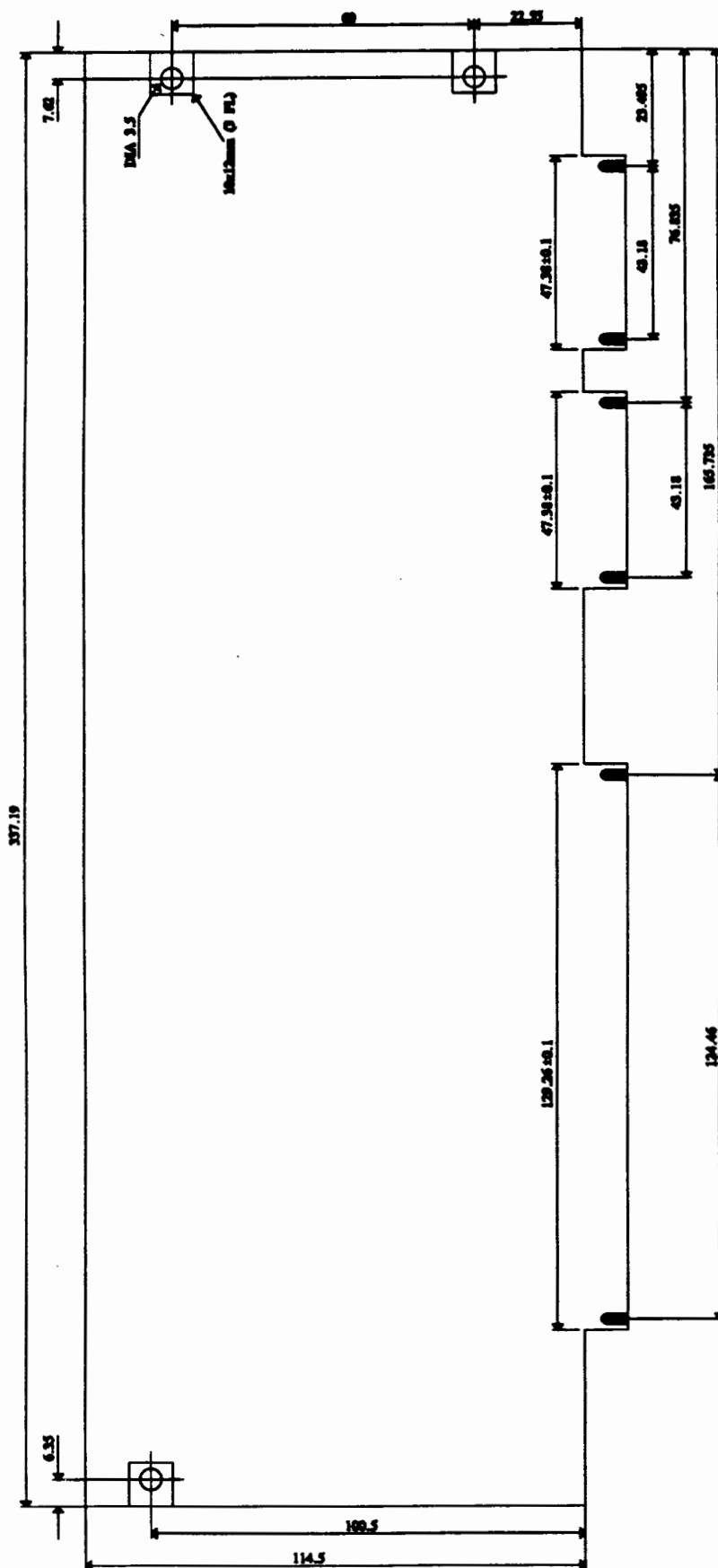
-Jeremy Bernstein

This section covers the various mechanical details of Zorro III cards. Note that these specifications are considered preliminary.



7.1 Basic Zorro III PIC

This drawing shows the basic Zorro III PIC. All of the dimensions are in millimeters.



7.3 PIC with Video Option

This drawing shows the basic Zorro III PIC, with both Zorro III and the Amiga Video Slot fingers specified. All of the dimensions are in millimeters.

This form factor is for the A3000 only. On the A4000, the video slot has been extended. See chapter 16 for a form factor specification of an A4000 PIC with video option.

CHAPTER 8

AUTOCONFIG®

"The goal of all inanimate objects is to resist man and ultimately defeat him."

-Russell Baker

8.1 The AUTOCONFIG® Mechanism

The AUTOCONFIG® mechanism used for the Zorro III bus is an extension of the original Zorro II configuration mechanism. The main reason for this is that the Zorro II mechanism works so well, there was little need to change anything. The changes are simply support for new hardware features on the Zorro III bus.

Amiga autoconfiguration is surprisingly simple. When an Amiga powers up or resets, every card in the system goes to its unconfigured state. At this point, the most important signals in the system are /CFGIN_N and /CFGOUT_N. As long as a card's /CFGIN_N line is negated, that card sits quietly and does nothing on the bus (though memory cards should continue to refresh even through reset, and any local board activities that don't concern the bus may take place after /RESET is negated). As part of the unconfigured state, /CFGOUT_N is negated by the PIC immediately on reset.

The configuration process begins when a card's /CFGIN_N line is asserted, either by the backplane, if it's the first slot, or via the configuration chain, if it's a later card. The configuration chain simply ensures that only one unconfigured card will see an asserted /CFGIN_N at one time. An unconfigured card that sees its /CFGIN_N line asserted will respond to a block of memory called *configuration space*. In this block, the PIC will assert a set of read-only

registers, followed by a set of write-only registers (the read-only registers are also known as AUTOCONFIG® ROM). Starting at the base of this block, the read registers describe the device's size, type, and other requirements. The operating system reads these, and based on them, decides what should be written to the board. Some write information is optional, but a board will always be assigned a base address or be told to shut up. The act of writing the final bit of base address, or writing anything to a shutup address, will cause the PIC to assert its /CFGOUTN, enabling the next board in the configuration chain.

The Zorro II configuration space is the 64K memory block \$00E8xxxx, which of course is driven with 16 bit Zorro II cycles; all Zorro II cards configure there. The Zorro III configuration space is the 64K memory block beginning at \$FF00xxxx, which is always driven with 32 bit Zorro III cycles (PICs need only decode A31-A24 during configuration). A Zorro III PIC can configure in Zorro II or Zorro III configuration space, at the designer's discretion, but not both at once. All read registers physically return only the top 4 bits of data, on D31-D28 for either bus mode. Write registers are written to support nybble, byte, and word registers for the same register, again based on what works best in hardware. This design attempts to map into real hardware as simply as possible. Every AUTOCONFIG® register is logically considered to be 8 bits wide; the 8 bits actually being nybbles from two paired addresses.

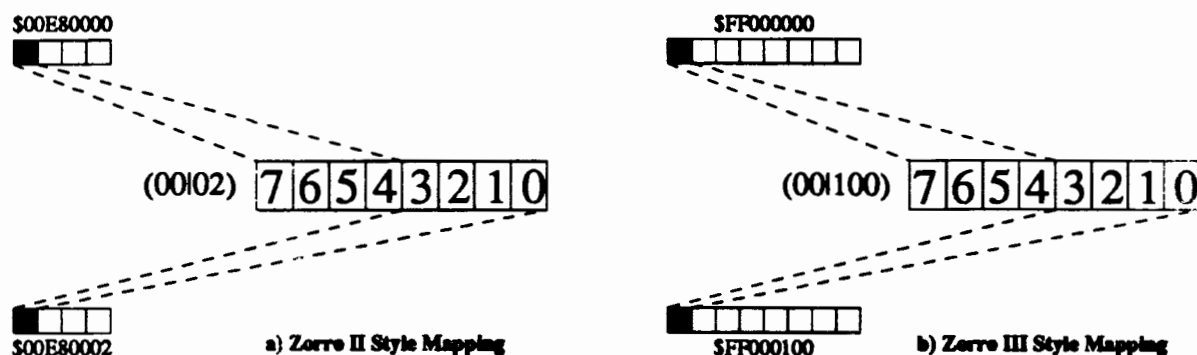


Figure 8-1: Configuration Register Mapping

The register mappings for the two different blocks are shown in *Figure 8-1*. All the bit patterns mentioned in the following sections are logical values. To avoid ambiguity, all registers are referred to by the number of the first register in the pair, since the first pair member is the same for both mapping schemes. In the actual implementation of these registers, all read registers except for the 00 register are physically complemented; eg, the logical value of register 3C is always 0, which means in hardware, the upper nybbles of locations \$00E8003C and \$00E8003E, or \$FF00003C and \$FF00013C, both return all 1's.

8.2 Register Bit Assignments

The actual register assignments are below. Most of the registers are the same as for the Zorro II bus, but are included here anyway for completeness. The Amiga OS software names for these registers in the ExpansionRom or ExpansionControl structures are included.

Reg Z2 Z3 Bit

00 **02 100 7,6**
 (or_Type)

These bits encode the PIC type:

00	Reserved
01	Reserved
10	Zorro III
11	Zorro II

- 5 If this bit is set, the PIC's memory will be linked into the system free pool. The Zorro III register 08 may modify the size of the linked memory.
- 4 Setting this bit tells the OS to read an autoboot ROM.
- 3 This bit is set to indicate that the next board is related to this one; often logically separate PICs are physically located on the same card.
- 2-0 These bits indicate the configuration size of the PIC. This size can be modified for the Zorro III cards by the size extension bit, which is the new meaning of bit 5 in register 08.

Bits	Unextended	Extended
000	8 megabytes	16 megabytes
001	64 kilobytes	32 megabytes
010	128 kilobytes	64 megabytes
011	256 kilobytes	128 megabytes
100	512 kilobytes	256 megabytes
101	1 megabyte	512 megabytes
110	2 megabytes	1 gigabyte
111	4 megabytes	RESERVED

04 **06 104 7-0**
 (or_Product)

The device's product number, which is completely up to the manufacturer. This is generally unique between different products, to help in identification of system cards, and it must be unique between devices using the automatic driver binding features.

08 **0A 108 7**
 (or_Flags)

This was originally an indicator to place the card in the 8 megabyte Zorro II space, when set, or anywhere it'll fit, if cleared. Under the Zorro III spec, this is set to indicate that the board is basically a memory device, cleared to indicate that the board is basically an I/O device.

- 6 This bit is set to indicate that the board can't be shut up by software, cleared to indicate that the board can be shut up.
- 5 This is the size extension bit. If cleared, the size bits in register 00 mean the same as under Zorro II, if set, the size bits indicate a new size. The

most common new Zorro III sizes are the smaller ones; all new sized cards get aligned on their natural boundaries.

- 4 Reserved, must be 1 for all Zorro III cards.
- 3-0 These bits indicate a board's sub-size; the amount of memory actually required by a PIC. For memory boards that auto-link, this is the actual amount of memory that will be linked into the system free memory pool. A memory card, with memory starting at the base address, can be automatically sized by the Operating System. This sub-size option is intended to support cards with variable setups without requiring variable physical configuration capability on such cards. It also may greatly simplify a Zorro III design, since 16 megabyte cards and up can be designed with a single latch and comparator for base address matching, while 8 megabyte and smaller PICs require large latch/comparator circuits not available in standard TTL packages.

Bits	Encoding
0000	Logical size matches physical size
0001	Automatically sized by the Operating System
0010	64 kilobytes
0011	128 kilobytes
0100	256 kilobytes
0101	512 kilobytes
0110	1 megabyte
0111	2 megabytes
1000	4 megabytes
1001	6 megabytes
1010	8 megabytes
1011	10 megabytes
1100	12 megabytes
1101	14 megabytes
1110	Reserved
1111	Reserved

For boards that wish to be automatically sized by the operating system, a few rules apply. The memory is sized in 512K increments, and grows from the base address upward. Memory wraps are detected, but the design must insure that its data bus doesn't float when the sizing routine addresses memory locations that aren't physically present on the board; data bus pullups or pulldowns are recommended. This feature is designed to allow boards to be easily upgraded with additional or increased density memory without the need for memory configuration jumpers.

Reg Z2 Z3 Bit

0C	0E 10C 7-0 (or_Reserved03)	Reserved, must be 0.
10	12 110 7-0	Manufacturer's number, high byte.
14	16 114 7-0 (or_Manufacturer)	Manufacturer's number, low bytes. These are unique, and can only be assigned by Commodore.
18	1A 118 7-0	Optional serial number, byte 0 (msb)
1C	1E 11C 7-0	Optional serial number, byte 1
20	22 120 7-0	Optional serial number, byte 2
24	26 124 7-0 (or_SerialNumber)	Optional serial number, byte 3 (lsb) This is for the manufacturer's use and can contain anything at all. The main intent is to allow a manufacturer to uniquely identify individual cards, but it can certainly be used for revision information or other data.
28	2A 128 7-0	Optional ROM vector, high byte.
2C	2E 12C 7-0 (or_InkDiagVec)	Optional ROM vector, low byte. If the ROM address valid bit (bit 4 of register (0002)) is set, these two registers provide the sixteen bit offset from the board's base at which the start of the ROM code is located. If the ROM address valid bit is cleared, these registers are ignored.
30	32 130 7-0 (or_Reserved0c)	Reserved, must be 0. Unsupported base register reset register under Zorro II*.
34	36 134 7-0 (or_Reserved0d)	Reserved, must be 0.
38	3A 138 7-0 (or_Reserved0e)	Reserved, must be 0.
3C	3E 13C 7-0 (or_Reserved0f)	Reserved, must be 0.
40	42 140 7-0 (or_Interrupt)	Reserved, must be 0. Unsupported control state register under Zorro II*.
44	46 144 7-0	High order base address register, write only.
48	4A 148 7-0 (or_Z3_HighByte) (or_BaseAddress)	Low order base address register, write only. The high order register takes bits 31-24 of the board's configured address, the low ordered register takes bits 23-16. For Zorro III boards configured in the Zorro II space, the configuration address is written both nybble and byte wide, with the ordering:

* The original Zorro specifications called for a few registers, like these, that remained active after configuration. Support for this is impossible, since the configuration registers generally disappear when a board is configured, and absolutely must move out of the \$00E8xxxx space. So since these couldn't really be implemented in hardware, system software has never supported them. They're included here for historical purposes.

Reg Z2 Z3 Bit

Reg	Nybble	Byte
46	A27-A24	N/A
44	A31-A28	A31-A24
4A	A19-A16	N/A
48	A23-A20	A23-A16

Note that writing to register 48 actually configures the board for both Zorro II and Zorro III boards in the Zorro II configuration block. For Zorro III PICs in the Zorro III configuration block, the action is slightly different. The software will actually write the configuration as byte and word wide accesses:

Reg	Byte	Word
48	A23-A16	N/A
44	A31-A24	A31-A16

The actual configuration takes place when register 44 is written, thus supporting any physical size of configuration register.

4C	4E 14C 7-0 (ec_Shutup)	Shut up register, write only. Anything written to 4C will cause a board that supports shut-up to completely disappear until the next reset.
50	52 150 7-0	Reserved, must be 0.
54	56 154 7-0	Reserved, must be 0.
58	5A 158 7-0	Reserved, must be 0.
5C	5E 15C 7-0	Reserved, must be 0.
60	62 160 7-0	Reserved, must be 0.
64	66 164 7-0	Reserved, must be 0.
68	6A 168 7-0	Reserved, must be 0.
6C	6E 16C 7-0	Reserved, must be 0.
70	72 170 7-0	Reserved, must be 0.
74	76 174 7-0	Reserved, must be 0.
78	7A 178 7-0	Reserved, must be 0.
7C	7E 17C 7-0	Reserved, must be 0.

CHAPTER 9

ZORRO III SIGNAL NAMES

"I have been given the freedom to do as I see fit."

-REM

The Zorro III Bus signals vary based on the particular bus mode in effect. This table lists each physical pin by physical name, and then by the logical names for Zorro II mode, Zorro III mode, address phase, and Zorro III data mode, data phase.

Pin No.	Physical Name	Zorro II Name	Zorro III Address Phase	Zorro III Data Phase
1	Ground	Ground	Ground	Ground
2	Ground	Ground	Ground	Ground
3	Ground	Ground	Ground	Ground
4	Ground	Ground	Ground	Ground
5	+5VDC	+5VDC	+5VDC	+5VDC
6	+5VDC	+5VDC	+5VDC	+5VDC
7	/OWN	/OWN	/OWN	/OWN
8	-5VDC	-5VDC	-5VDC	-5VDC
9	/SLAVEN	/SLAVEN	/SLAVEN	/SLAVEN
10	+12VDC	+12VDC	+12VDC	+12VDC
11	/CFGOUT _N	/CFGOUT _N	/CFGOUT _N	/CFGOUT _N
12	/CFGIN _N	/CFGIN _N	/CFGIN _N	/CFGIN _N
13	Ground	Ground	Ground	Ground
14	/C3	/C3 Clock	/C3 Clock	/C3 Clock
15	CDAC	CDAC Clock	CDAC Clock	CDAC Clock
16	/C1	/C1 Clock	/C1 Clock	/C1 Clock

Pin No.	Physical Name	Zorro II Name	Zorro III Address Phase	Zorro III Data Phase
17	/CINH	/OVR	/CINH	/CINH
18	/MTCR	XRDY	/MTCR	/MTCR
19	/INT ₂	/INT ₂	/INT ₂	/INT ₂
20	-12VDC	-12VDC	-12VDC	-12VDC
21	A ₅	A ₅	A ₅	A ₅
22	/INT ₆	/INT ₆	/INT ₆	/INT ₆
23	A ₆	A ₆	A ₆	A ₆
24	A ₄	A ₄	A ₄	A ₄
25	Ground	Ground	Ground	Ground
26	A ₃	A ₃	A ₃	A ₃
27	A ₂	A ₂	A ₂	A ₂
28	A ₇	A ₇	A ₇	A ₇
29	/LOCK	A ₁	/LOCK	/LOCK
30	AD ₈	A ₈	A ₈	D ₀
31	FC ₀	FC ₀	FC ₀	FC ₀
32	AD ₉	A ₉	A ₉	D ₁
33	FC ₁	FC ₁	FC ₁	FC ₁
34	AD ₁₀	A ₁₀	A ₁₀	D ₂
35	FC ₂	FC ₂	FC ₂	FC ₂
36	AD ₁₁	A ₁₁	A ₁₁	D ₃
37	Ground	Ground	Ground	Ground
38	AD ₁₂	A ₁₂	A ₁₂	D ₄
39	AD ₁₃	A ₁₃	A ₁₃	D ₅
40	Reserved	(/EINT ₇)	Reserved	Reserved
41	AD ₁₄	A ₁₄	A ₁₄	D ₆
42	Reserved	(/EINT ₅)	Reserved	Reserved
43	AD ₁₅	A ₁₅	A ₁₅	D ₇
44	Reserved	(/EINT ₄)	Reserved	Reserved
45	AD ₁₆	A ₁₆	A ₁₆	D ₈
46	/BERR	/BERR	/BERR	/BERR
47	AD ₁₇	A ₁₇	A ₁₇	D ₉
48	/MTACK	(/VPA)	/MTACK	/MTACK
49	Ground	Ground	Ground	Ground
50	E Clock	E Clock	E Clock	E Clock
51	/DS ₀	(/VMA)	/DS ₀	/DS ₀
52	AD ₁₈	A ₁₈	A ₁₈	D ₁₀
53	/RESET	/RST	/RESET	/RESET
54	AD ₁₉	A ₁₉	A ₁₉	D ₁₁
55	/HLT	/HLT	/HLT	/HLT
56	AD ₂₀	A ₂₀	A ₂₀	D ₁₂
57	AD ₂₂	A ₂₂	A ₂₂	D ₁₄
58	AD ₂₁	A ₂₁	A ₂₁	D ₁₃
59	AD ₂₃	A ₂₃	A ₂₃	D ₁₅

Pin No.	Physical Name	Zorro II Name	Zorro III Address Phase	Zorro III Data Phase
60	/BRN	/BRN	/BRN	/BRN
61	Ground	Ground	Ground	Ground
62	/BGACK	/BGACK	/BGACK	/BGACK
63	AD31	D15	A31	D31
64	/BGN	/BGN	/BGN	/BGN
65	AD30	D14	A30	D30
66	/DTACK	/DTACK	/DTACK	/DTACK
67	AD29	D13	A29	D29
68	READ	READ	READ	READ
69	AD28	D12	A28	D28
70	/DS2	/LDS	/DS2	/DS2
71	AD27	D11	A27	D27
72	/DS3	/UDS	/DS3	/DS3
73	Ground	Ground	Ground	Ground
74	/CCS	/AS	/CCS	/CCS
75	SD0	D0	Reserved	D16
76	AD26	D10	A26	D26
77	SD1	D1	Reserved	D17
78	AD25	D9	A25	D25
79	SD2	D2	Reserved	D18
80	AD24	D8	A24	D24
81	SD3	D3	Reserved	D19
82	SD7	D7	Reserved	D23
83	SD4	D4	Reserved	D20
84	SD6	D6	Reserved	D22
85	Ground	Ground	Ground	Ground
86	SD5	D5	Reserved	D21
87	Ground	Ground	Ground	Ground
88	Ground	Ground	Ground	Ground
89	Ground	Ground	Ground	Ground
90	Ground	Ground	Ground	Ground
91	SenseZ3	Ground	SenseZ3	SenseZ3
92	7M	E7M	7M	7M
93	DOE	DOE	DOE	DOE
94	/IORST	/BUSRST	/IORST	/IORST
95	/BCLR	/GBG	/BCLR	/BCLR
96	Reserved	(/EINT1)	Reserved	Reserved
97	/FCS	No Connect	/FCS	/FCS
98	/DS1	No Connect	/DS1	/DS1
99	Ground	Ground	Ground	Ground
100	Ground	Ground	Ground	Ground

CHAPTER 10

INTRODUCTION TO THE LOCAL BUS

"What works for me might work for you"

-Jimmy Buffet

This article describes the 200-pin Local Bus Expansion Slot (also known as the Coprocessor Slot) of the A4000 and A3000 Amiga models. This expansion slot is designed to provide high speed access to the Amiga's local, or 68030, bus. This slot is intended for high speed expansion devices that are generally very specific to the 68030 bus or need direct access to the local bus for other reasons. Such devices include alternate 680x0 family processors, cache memory boards, high speed bursting RAM expansion, and similar things.

10.1 Intended Audience

The information presented here is for hardware engineers interested in designing cards for the Local Bus Slot. A good level of microcomputer systems design knowledge is necessary to get much meaning out of these pages. Especially important is familiarity with the 68030 processor hardware conventions, as most of the Local Bus Slot is based directly on the 68030 processor bus. These conventions are described in the *MC68030 User's Manual* by Motorola (3rd Edition, Prentice-Hall, ISBN 0-13-566423-3).

10.2 Why a Local Bus Slot?

The local bus slot was originally introduced on the Amiga 2000 and has served its intended purpose quite well on that system. On the A3000 and A4000 the slot has been expanded from 86 pins to 200 pins but serves the same basic purpose. Its main use is to allow the addition of a single, very tightly coupled expansion device, typically a high speed CPU, cache, or memory board. An alternate 680x0 family device, such as a 68040 processor, needs access to every 68030 signal in order to properly replace the 68030 processor as a host for AmigaOS or UNIX. Cache memories or high-performance Fast memory need direct access to

the 68030 bus to run as tightly coupled as possible to the 68030 processor on the motherboard. Most other types of expansion devices should be designed as Zorro II or Zorro III expansion cards.

In any system design, you can make good arguments for a general purpose expansion bus, and good arguments for an extendable local bus. The Amiga philosophy is that both of these approaches are correct, and serve complementary needs. The kinds of devices that would work well in the Local Bus Slot are by their very nature tightly coupled to the system bus, which is of course based on the 68030 bus. Such devices are not expected to work in future Amiga systems, which could easily have completely different local buses, and therefore, different local bus slots. Also, it's impractical to provide a large number of such slots, since the local bus itself has tight electrical limits on expandability. So a single local bus slot is provided.

10.3 Why an Expansion Bus Slot?

Most add-on cards for an A3000 or A4000 belong in a Zorro III Expansion Bus Slot. First of all, since there is only one Local Bus Slot, it stands to reason that only one such device can be added to any system, while four Zorro III Expansion Bus slots are available on the A3000 and A4000 models. The Zorro III bus doesn't permit the same degree of tight coupling that the Local Bus Slot does, so it is not capable of supporting cache or other zero wait-state memory, and it can't support a direct-replacement 68040. It does permit reasonable speeds, interrupts, bus locking, etc. so it is the place for high performance I/O devices, moderate speed add-on memory boards, processor devices such as DSP, video, or RISC devices that coexist with the main processor, etc.

And of course, devices that are happy as slower 16-bit peripherals can be implemented as Zorro II cards and have the advantage of working in all Amiga computers (including the A500 and A1000 with the proper 3rd party bus adaptor or backplane). Details on the Zorro III bus are available in The Zorro III Expansion Bus Specification in chapters 1-9 of this document and also listed in Appendix K (p. 383) of the *Amiga Hardware Reference Manual*, 3rd Edition (ISBN 0-201-56776-8).

CHAPTER 11

FUNCTIONALITY AND DESIGN GUIDELINES

"Time and distance are out of place here"

-REM

The Local Bus Expansion Slot provides signals to implement both slave and master devices. Memory devices, including cache, are bus slaves, while CPU devices such as 680x0 accelerator boards are bus masters. Any card may, at times, be either slave or master, and in a few cases, both at once.

11.1 Slave Devices

A slave device is a device that responds to the current local bus master. The local connector provides direct access to all local bus signals, which include all 68030 signals, plus a few additional Amiga-specific lines to allow a Local Bus Slot device to control the Amiga's Local Bus.

Local Bus Slot slaves are not autoconfigured like a Zorro III bus device, but instead are fixed in the address range from \$08000000 to \$0FFFFFFF. The Local Bus Slot controller, the Fat Gary chip, provides a decode of this space on the signal /RAMSLOT which is valid at address time. This signal can be ignored and the address decoded by logic on the board if speed is an issue.

Local slave devices should also support the signal /CIIN if they contain uncachable data. The Amiga OS expects anything mapped starting at \$08000000 to be memory, and in fact, the fastest memory in the A3000 and A4000 systems. The OS will automatically size and link in any memory it finds here, and place it in the system as the highest priority memory available. To support control registers or other memory mapped resources on a Local Bus Card, locate them at least 512K above the \$08000000 base, and the OS will ignore them.

A signal named /WAIT is provided for cache support. Asserting this signal will disable address decoding of onboard Fast RAM by the RAMSEY chip and Zorro II/III bus accesses by the BUSTER chip. Constraints imposed by the 68030 allow only 18ns to determine a cache hit. It is often more feasible to assert /STERM before knowing whether the cycle is a cache hit or cache miss and rerunning the cycle via /HALT and /BERR if it is a miss. To achieve this functionality any decoding of the first cycle by RAMSEY or BUSTER must be disabled by asserting /WAIT less than 10ns after address valid. If the cycle is determined to be a cache miss, a rerun is initiated and wait deasserted for the secondary cycle. Assertion of /WAIT will keep /STERM, /CBACK, etc. tristated by BUSTER or RAMSEY and may be controlled by the cache control logic.

11.2 Master Devices

A Master Device is, of course, a device which masters the local bus, replacing the functions of the 68030 during its period of mastership. Bus mastership may be accomplished two ways depending on the desired functionality. The first mode, called primary mastership, totally disables the motherboard 68030 and its arbitration logic, essentially replacing the 68030 with the local slot device. Such a device takes over full arbitration responsibility for the whole system, and must service all interrupts.

The second mode, called secondary mastership, allows the on-board 68030 and the local bus accelerator board to share the bus, permitting multiprocessor capabilities or more traditional DMA from the local bus board. This protocol permits very fast switching between the local slot master and the motherboard 68030. In this mode, the 68030 is still responsible for bus arbitration.

11.2.1 Primary Bus Mastership

The primary bus mastership, or arbitration takeover mode, requires less logic to implement and may be preferred in most implementations. In the absence of multiprocessing software support, this is the mode of choice, and corresponds to the way in which most A2000 local slot cards worked.

The local bus card asserts /CBR at power on to the motherboard which in turn asserts /BR to the motherboard 68030. Upon receiving /BG30 from the motherboard the local card asserts /BOSS. Logic on the motherboard uses /BOSS to force /BGACK30 low to the 68030 only and not the shared local bus /BGACK. In addition the assertion of /BOSS tristates /BG on the motherboard and in turn the local card should untristate and source its /BG.

The local card is now the default bus master and arbiter -- it must provide arbitration for the local bus, based on the 68030 bus arbitration rules. The onboard 68030 is bus arbitrated away and never regains the bus. PAL equations to implement this are given in Figure 11-1. All PAL equations are active high and should be inverted in the output stage of the PAL for active low assertion.

```

/* Always assert Coprocessor Bus Request. */
CBR      =      'b'1;

/* The Boss Signal. "poweron_reset" is a signal sourced by the local card and
   is asserted for a few hundred nanoseconds after poweron. This clears the
   feedback path on the pal and may be generated by an RC network which is
   slewrate cleaned by a schmitt trigger device. */
BOSS      =      BG30 & BOSS & !poweron_reset;

/* The Grant Signal. "local_card_bg" is sourced by the local card in
   compliance to the operation of arbitration defined in the 68030 users
   manual. */
BG        =      local_card_bg;

BG.oe     =      BOSS;

```

Figure 11-1: Primary Mode Takeover PAL Equations

Actual timing for takeover mode is not given since all signals are inherently asynchronous. The untristating of /BG should be later than the tristating of /BG on the motherboard to minimize contention on that signal.

11.2.2 Secondary Bus Mastership

The secondary bus acquisition mode uses the 68030 arbiter to provide cycle arbitration between the Local Bus Slot card and the motherboard DMA. Since the 68030 provides only a single bus request input, a scheme referred to as fast arbitration is used between the local card and all other DMA sources, which are controlled via the BUSTER chip. Bus request is an open collector line which is time multiplexed between the local card and BUSTER. On a positive edge of CPUCLK, BUSTER will assert /BR if it requires the bus and /BR is not already asserted by the local bus card. On the negative transition of CPUCLK, the local bus card may assert /BR if it is not already asserted by BUSTER. This scheme allows both masters to share a single bus request and also requires only 20ns to resolve the master arbitration. This mode is very efficient but forces the local card to use high speed logic since the time between clock edges is so short.

It is strongly suggested that the above logic be incorporated in a 7.5ns or faster registered PAL connected to a F38 open collector device. Clock skew between the clock to this PAL and CPUCLK is critical and it is advised that the local card generate and source clocks to the motherboard, especially if CPUCLK is needed elsewhere on the card; load on CPUCLK at the Local Bus Slot must be kept to a minimum to avoid mucking with the local bus timing. Skew between CPUCLK and the clock driving this PAL should be less than 2ns. The PAL equations for /BR are given in Figure 11-2.

After receiving /BG from the motherboard 68030 the local card drives /BGACK (open collector) and assumes mastership of the bus. It may keep the bus for multiple cycles but should not hog the bus for extended periods unless it relinquishes the bus when DMA request is asserted by BUSTER. Hogging the bus may cause adverse operation of the system. Be aware that the

```

/* This is the BR_LOCAL signal, which is an active output fed into a 74f38, or
other open collector equivalent buffer, to produce /BR. /BR is the raw bus
request from the local bus connector (keep this trace short on the local
card). WANTBUS is the request from the local card which deasserts after it
sees BR_LOCAL asserted. BGACK_LOCAL is /BGACK asserted by the local card.
RESET is any reset signal used to prevent poweron latchup of BR_LOCAL. */

```

```

BR_LOCAL.d =    !BR & !BR_LOCAL & WANTBUS
               #    BR_LOCAL & !BGACK_LOCAL & !RESET;

```

Figure 11-2: Secondary Mode Takeover PAL Equations

local bus signals must be tristated (/AS, /DSACK, Address, Data, etc.) prior to deasserting /BGACK. In addition the local card must not drive the bus or /BGACK until /AS, /DSACK, /BERR, /HALT, etc. have deasserted. In other words, standard 680x0 bus arbitration rules apply.

It is generally assumed that any secondary bus master mode Local Bus Slot device will attempt to prevent undue bus hogging at the design level. This implies either a coprocessor device of some kind that makes only periodic requests of the bus, or a CPU subsystem that contains its own local cache or memory. Any device that requires the mastering of the local bus for very long periods of time should be a primary mode bus master.

It is extremely important that the bus master timing from the local card emulate operation of a 25MHz or 16MHz 68030 chip exactly as defined by the *MC68030 User's Manual* (ISBN 0-13-566423-3). The bus control chips currently incorporate burst cycles and cache coherency signals (/CIIN) and future enhancements may incorporate rerun cycles. Signals received by the local card from the motherboard provide only the minimum setup and hold required by a 68030. Do not assume for example that data setup from the motherboard will not significantly change, i.e., data setup from Fast RAM on subsequent cycles of a burst is much less than a typical non-burst cycle.

In addition to the 68030 specifications, the Amiga local bus adds one additional constraint. Slave devices should hold data through the end of cycle, regardless of whether the cycle is terminated by /STERM or the /DSACK lines. In other words, only the current bus master really knows when a cycle has completed. Obviously, burst cycles are an exception to this; they follow the standard 68030 rules in dealing with data hold times.

11.3 Clock generation

The A3000 and A4000 motherboards provide links to disable generation of CPUCLK and CLK90. This allows the Local Bus Slot card to maintain better clock skew relationships between its own logic and that of the motherboard, and is especially important if the local bus card depends upon being synchronous to the motherboard clocks. Just as with the A2000 local bus slot, both synchronous and asynchronous designs are possible, but synchronous designs are much more timing-critical than on the A2000.

If a local bus card drives the clock lines the appropriate jumpers must be moved on the motherboard. CLK90 must be a clock 90° out of phase from CPUCLK. It is typically generated from a 5 tap 25ns delay line where CLK90 is the 10ns tap when running the system at 25Mhz and CLK90 is the 15ns tap when at 16Mhz. Note that these clocks are fed through a 74f08 to provide clocking to the motherboard. If the skew generated by the f08 is unacceptable (as in fast arbitration) the socketed f08 may be removed and replaced by a header which shorts the appropriate inputs to outputs, though it is the responsibility of the local bus card at that point to make sure acceptable versions of CPUCLK and CLK90 exist everywhere on the motherboard. Again careful layout of the clock circuitry is essential for reliable operation.

11.4 Local Bus Design Criteria

Any design which plugs into the Local Bus Slot connector must comply to some basic design rules.

- ☐ Any design which plugs into the Local Bus Slot connector must comply to some basic design rules.
- ☐ Due to inductance in the 200-pin connector to VCC and GND it is very important to provide ample bypass capacitance in order to maintain good DC VCC and GND levels.
- ☐ All signals from this connector are unbuffered and should not be heavily loaded. A good rule is 2 TTL loads. In addition receivers and drivers should be located near the connector and any connector signal should not run over 4 inches in length from the connector before entering or leaving a driver or receiver.
- ☐ Clock generation is especially critical. Keep traces short, ECL routing rules should be followed if possible. Fan out multiple clocks from a single die to minimize loading per clock. Light damping resistors minimize radiation but cause clock distortion, so tune the values carefully.
- ☐ Keep in mind that current draw in the A3000 and A4000 is tight so use CMOS and powerdown DRAM modes when possible. New FCT devices use significantly lower current than F and run faster. A local bus card should draw no more than 2 Amps @ 5 VDC.
- ☐ Be aware of heat dissipation issues especially on very high speed microprocessors.
- ☐ Noise test local bus cards and ensure good AC signal quality since a nasty signal will get nastier after passing through an inductive connector to the motherboard. And keep in mind that any noise a local card injects into the system will make the entire system less reliable.

- ❑ Local bus card mounting holes are plated through to ground on the A3000 and A4000 motherboard and provide an additional low inductance path to ground. Use this path to minimize ground bounce relative to the motherboard.

CHAPTER 12

LOCAL BUS SIGNAL DESCRIPTIONS

*"There's a name for it
And names make all the difference in the world"*
-David Byrne

The signals on the Local Bus Slot can be broken down into several categories. Some of these are in common with the 68030, some are specific to this slot. The pinout is listed at the end of this chapter.

12.1 Power Connections

These signals provide digital supply levels to the local bus card. There are quite a few of both levels on the physical connector; generally at least one for every two or three signal pins.

Digital Ground (GND)

This is the digital supply ground used by all digital devices in the system. The local bus card gets GND through its mounting posts as well as the connector pins.

Digital Supply (+5VDC)

This is the digital supply. This is specified as +5VDC +/- 5%. The system power budget allocates up to 2 Amps for this slot.

12.2 System Initialization

These signals are driven by the motherboard logic to initialize the system; local bus cards should listen as appropriate.

/RESET

This is an open-collector signal driven by the system reset logic or, indirectly, any CPU device that needs to reset the I/O subsystem. Local bus cards generally don't use this, though it can be used to reset I/O devices or hold the main system in reset if necessary.

/FPURST

This active input is driven by the system reset logic to indicate the full CPU register reset condition.

/CPURST

This open-collector signal is driven by the system reset logic to indicate a full CPU register reset condition to the CPU, and can be driven by the CPU to cause an I/O reset in the rest of the system.

12.3 68030 Signals

All of these signals are directly connected to the 68030, and more information on them is available in the *68030 User's Manual*. Most of these must be driven or sampled by any local slot DMA device.

Address Bus (A31 - A0)

The 32-bit processor address bus, driven by the bus master, tristated by inactive masters.

Data Bus (D31 - D0)

The 32-bit processor data bus, driven by the bus master for writes, the slave for reads. This is tristated when outside of a bus cycle (/AS is negated).

Function Codes (FC2 - FC0)

An address bus extension, driven by the bus master, tristated by inactive masters. Most slaves respond only to FC0 + FC1.

Bus Size (SIZ1 SIZ0)

Data bus size request, driven by the bus master, tristated by inactive masters.

Cycle Strokes (/AS, /DS)

/AS indicated the start of a bus cycle and valid addresses, /DS indicates valid data for write cycles. Both are driven by the bus master, tristated by inactive bus masters.

Read Indicator (R/W)

Driven by this bus master, this tristated signal is high to indicate a read, low to indicate a write.

Read-Modify-Write Cycle (/RMC)

This line is asserted by the bus master to effect a bus lock; while it is active, no bus arbitration takes place, and shared memory coprocessors (except Agnus) stay out of the memory involved in the transaction.

Other Strokes (/ECS, /OCS, /DBEN)

These are additional 68030 strokes that aren't used by the Amiga and therefore, don't have to be driven by a local bus master. They are provided here for the possible use of any slave device that wants them; see the *68030 User's Manual* for signal details.

Burst Control (/CBREQ, /CBACK)

The local bus slave drives /CBREQ to indicate that its capable of supporting a burst cycle. The local bus master responds with /CBACK to indicate that it can run a burst cycle.

Cache Control (/CIIN, /CIOUS)

The bus slave drives the /CIIN line to indicate that the currently addressed location is uncachable. The bus master drives /CIOUS to indicate that the current location is uncachable, based on MMU tables as well as /CIIN.

Cycle Termination (/STERM, /DSACK, /DSACK, /AVEC).

The bus slave drives either /STERM or one or more /DTACKs to normally terminate a cycles. The combination of DSACK lines indicates the bus port size; /STERM can only be generated by 32-bit-port slaves. The /AVEC line is driven by local bus logic to terminate an interrupt acknowledge cycle with an autovector rather than a device-supplied vector.

Interrupts (/IPL2 - /IPL0)

Encoded interrupt inputs. These are generally serviced only by the primary bus master, though other schemes are possible with the proper software support. They are inputs; they can't ever be driven by a slave or a master.

Interrupt Pending (/IPEND)

Driven by the 68030 to indicate that there is a pending interrupt to be serviced. A secondary bus master may use this as an indication to let the 68030 back onto the local bus, if the 68030 is handling the interrupts.

Exceptions (/HALT, /BERR)

/HALT driven alone causes the CPU to stop; generally this is used by single-stepping emulators. /HALT driven with /RESET indicates a full 68000 style reset, and is considered archaic on the A3000 and A4000 local bus. /BERR driven alone indicates some kind of bus error, generally a bus collision or timeout. /BERR and /HALT driven together indicate a bus retry.

12.4 Bus Arbitration Signals

These are the signals used to arbitrate the local bus in the ways previously described, supporting both primary and secondary bus masters.

Bus Requests (/BR, /CBR)

The both the /BR and /CBR line cause the bus to be requested from the 68030. The /BR line is for secondary bus masters, and it is a time multiplexed open collector line shared with a similar /BR output from the Buster chip. The /CBR line causes a request to go to the 68030, and once the primary arbitration is completed, the new primary master on the Local Bus Slot must deal with incoming /BR signals from Buster.

SCSI Bus Request (/SBR)

This line allows the local bus card to monitor when the SCSI devices wants the bus; this is primarily used as an indicator to secondary masters to give up the local bus.

Bus Grants (/BG, /BG30)

The /BG line is the main local bus grant signal. It is normally generated by the 68030, but when a primary master takes over, this line will tristate, allowing the primary master to drive /BG in response to an incoming /BR. The /BG30 line is the bus grant line coming directly from the 68030 chip.

Bus Grant Acknowledges (/BGACK, /BOSS)

The /BGACK signal is the main bus grant acknowledge, shared by Buster and the DMAC. Secondary masters drive /BGACK to acquire the local bus. The /BOSS signal is a private /BGACK-equivalent to the 68030, used by a primary bus master to acquire the bus from the 68030.

Bus Clear Request (/EBCLR)

This is a signal from the bus arbiter, indicating that some other bus master wants the local bus. This is generally used by a secondary bus master as an indicator of when to get off the bus.

12.5 Other Local Bus Signals

Local Slot Memory Decode (/RAMSLOT)

This is an address based chip select for the region of memory allocated to the local bus slot, \$08000000-\$0ffffff.

Emulator Mode (/EMUL)

This signal can be driven by local bus slot emulator devices to pull the /CDIS and /MMUDIS lines on the 68030, thereby disabling the cache and MMU for debugging purposes.

Cycle Wait (/WAIT)

This line is asserted by a bus monitoring device, such as a cache, to hold off cycle start by either the memory controller (RAMSEY) or expansion bus controller (BUSTER). This gives the device time to determine if it owns that address, retry the cycle, or anything else necessary to support cache and similar kinds of devices.

FPU Chip Select (/FPUCS)

This is a decode for the Coprocessor Device 1, the FPU, generated by the Gary Chip.

12.6 Clocks

This section details the Amiga system clocks available at the Local Bus Slot, the clocking alternatives available to a local bus device, and various clock control lines to facilitate this control.

System Clocks (CPUCLK, CLK90)

These are the main Amiga system clocks. CPUCLK is a 16MHz or 25MHz clock, depending on the system configuration, and is the main system, CPU, and FPU clock. CLK90 is CPUCLK shifted 90°.

External Clocks (EXTCLK, EXT90)

These clocks can be driven to replace the on-board clocks to the main system. Motherboard jumpers can be arranged to permit the use of these clocks.

12.7 Amiga 3000T Signals

These Local Bus Slot signals appear only in the tower version of the A3000, the A3000T.

System Clock Steal (DIS_CLKS)

This implements an alternate clock replacement method. When the DIS_CLKS line is driven high, the replacement clocks can be driven onto the local bus. This eliminates the need for any jumper adjustments to be made on the motherboard when a clock sourcing board is installed.

Replacement Clocks (ECPUCLK, ECPUCLKB, ECLK90, ECLK90A)

These are replacement main system clocks, and their 90° counterparts, that can be directly driven onto the local bus when DIS_CLKS is asserted.

CPU Clock Steal (DIS_CLK30, ECLK30)

This allows the 68030 clock to be driven from the local bus, again for skew reduction or other such tricks. When DIS_CLK30 is asserted, ECLK30 can be driven by local bus card logic.

12.8 Amiga 4000 Signals

These local bus slot signals appear in the A4000 version of the Amiga only.

Interrupt 6 Out (/INT6)

This is the shared level six interrupt line. It can be driven by a Local Bus Slot device to interrupt the host CPU.

Interrupt 2 Out (/INT2)

This is the shared level two interrupt line. It can be driven by a Local Bus Slot device to interrupt the host CPU.

SCSI (/SCSI)

Address decode from GARY for \$00DD0000 - \$00DD3FFF of user and supervisor space. Reserved.

DMA Enable (/DMAEN)

Turns on DMA counter in RAMSEY. Reserved.

12.9 Local Bus Connector Pinout

Here is a complete list of the pins and signals on the local bus connector. For a description of the signals see the preceding section. A more complete description of the standard 68030 inputs and outputs is available from the *68030's User Manual* (ISBN 0-13-566423-3).

The local bus connector is a two-piece, 200-pin, high-density connector. It is made by KEL. Be very careful with connection direction and pinout when doing board layouts.

Pin/ Signal Name	Pin/ Signal Name
1 /DSACK1	35 Ground
2 Ground	36 A [0]
3 Ground	37 A [9]
4 /HALT	38 Ground
5 R/W	39 Ground
6 Ground	40 A [1]
7 Ground	41 A [10]
8 /BGACK	42 Reserved (ECLK90A on A3000T)
9 /SBR	43 /INT6 (Unused in A3000/A3000T)
10 Ground	44 A [2]
11 Ground	45 A [11]
12 /AVEC	46 Reserved (ECLK90 on A3000T)
13 EXT90	47 Ground
14 +5 VDC	48 A [3]
15 +5 VDC	49 A [12]
16 /RAMSLOT	50 Ground
17 /BOSS	51 Ground
18 +5 VDC	52 A [4]
19 +5 VDC	53 A [13]
20 FC [0]	54 Reserved (ECPUCLKB on A3000T)
21 /STERM	55 /WAIT
22 +5 VDC	56 A [5]
23 +5 VDC	57 A [14]
24 FC [1]	58 Reserved (ECPUCLKA on A3000T)
25 /BR	59 Ground
26 +5 VDC	60 A [6]
27 +5 VDC	61 A [15]
28 /CBACK	62 Ground
29 /BERR	63 Ground
30 Reserved (DIS_CLKS on A3000T)	64 A [7]
31 /EMUL	65 A [16]
32 /CBREQ	66 /SCSI (Unused in A3000/A3000T)
33 A [8]	67 /DMAEN (Unused in A3000/A3000T)
34 Reserved (ECLK30 on A3000T)	68 A [24]

Pin/ Signal Name

69	A [17]
70	Reserved (DIS_CLK30 in A3000T)
71	Ground
72	A [25]
73	A [18]
74	Ground
75	Ground
76	A [26]
77	A [19]
78	Reserved
79	Reserved
80	A [27]
81	A [20]
82	/INT2 (Unused in A3000/A3000T)
83	Ground
84	A [28]
85	A [21]
86	Ground
87	Ground
88	A [29]
89	A [22]
90	Reserved
91	/DSACK0
92	A [30]
93	A [23]
94	+5 VDC
95	+5 VDC
96	A [31]
97	/DS
98	+5 VDC
99	+5 VDC
100	/ECS
101	/CIOUT
102	+5 VDC
103	+5 VDC
104	/DBEN
105	/BG
106	+5 VDC
107	+5 VDC
108	/RMC
109	/CPURST
110	/FPURST
111	Reserved

Pin/ Signal Name

113	/EBCLR
114	Reserved
115	Ground
116	/IPEND
117	/RESET
118	Ground
119	Ground
120	/IPL [0]
121	SIZ0
122	Ground
123	Ground
124	/IPL [1]
125	FC [2]
126	CLK90_EXP
127	Ground
128	/IPL [2]
129	SIZ1
130	Ground
131	Ground
132	/CIIN
133	/AS
134	/FPUCS
135	CPUCLK_EXP
136	/OCS
137	D [31]
138	Ground
139	Ground
140	D [15]
141	D [30]
142	Ground
143	Ground
144	D [14]
145	D [29]
146	Reserved
147	/CBR
148	D [13]
149	D [28]
150	Reserved
151	Ground
152	D [12]
153	D [27]
154	Ground
155	Ground
156	D [11]

Pin / Signal Name

157	D [26]
158	Reserved
159	/BG30
160	D [10]
161	D [25]
162	Reserved
163	Ground
164	D [9]
165	D [24]
166	Ground
167	Ground
168	D [8]
169	D [16]
170	Reserved
171	Reserved
172	D [0]
173	D [17]
174	+5 VDC
175	+5 VDC
176	D [1]
177	D [18]
178	+5 VDC
179	+5 VCD
180	D [2]
181	D [19]
182	+5 VDC
183	+5 VDC
184	D [3]
185	D [20]
186	+5 VDC
187	+5 VDC
188	D [4]
189	D [21]
190	Ground
191	Ground
192	D [5]
193	D [22]
194	Ground
195	Ground
196	D [6]
197	D [23]
198	Ground
199	Ground
200	D [7]

CHAPTER 13

LOCAL BUS FORM FACTORS

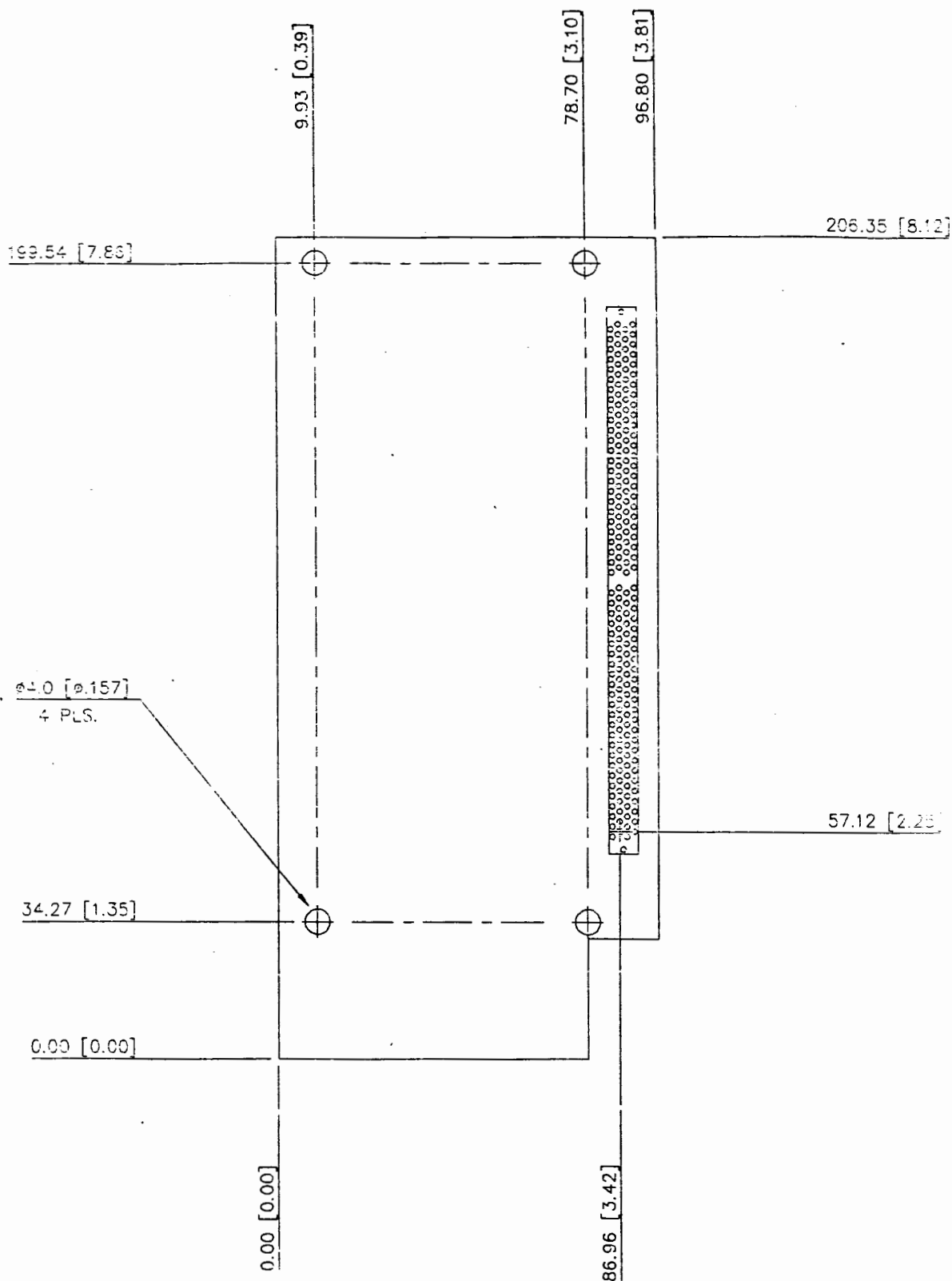
*"As you will see from my drawing, everything viewed through
them is reversed and appears in mirror image."*

-M. C. Escher

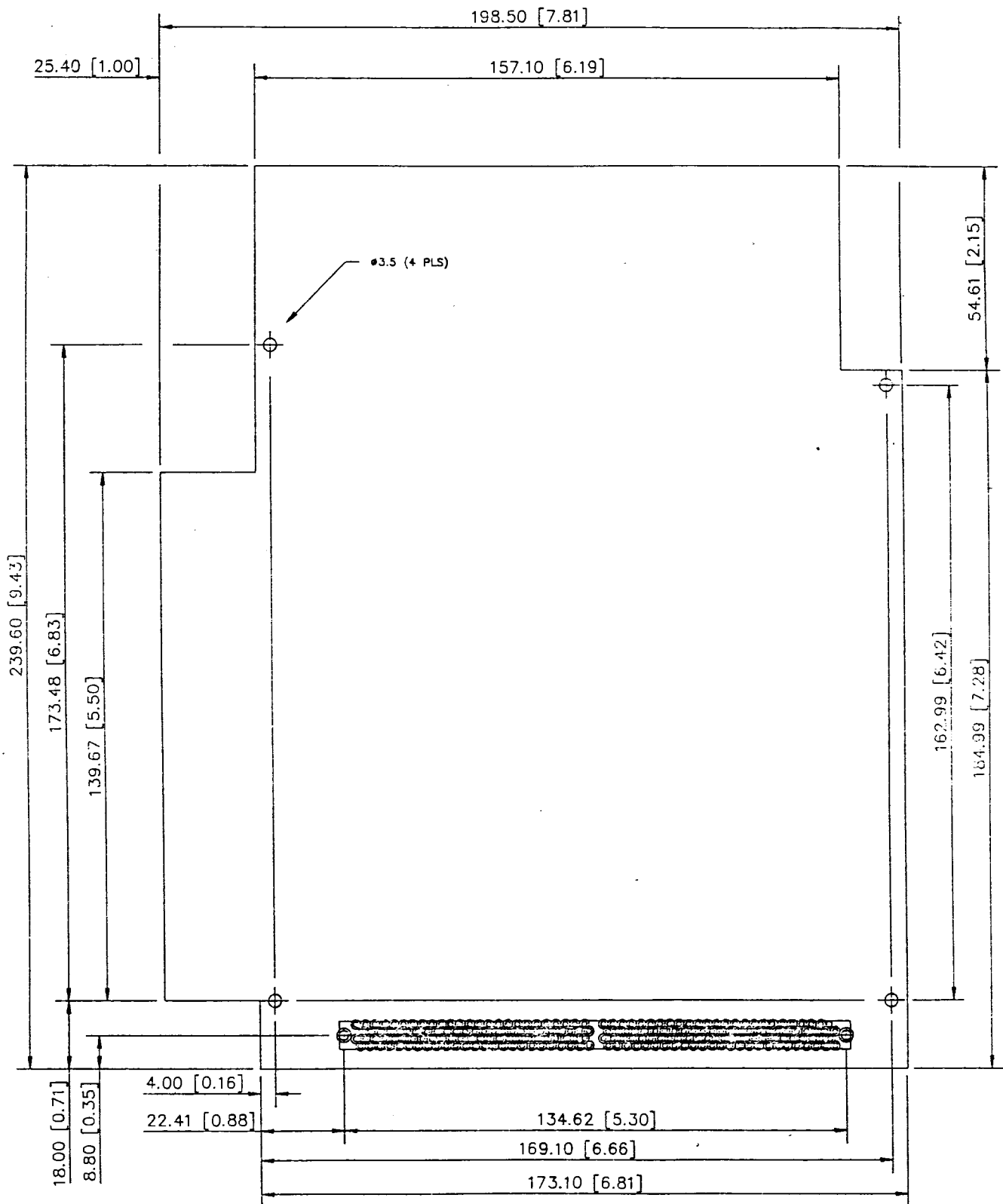
The form factors for a card that plugs into the local bus slot of the A4000 and A3000 are shown on the next two pages. The drawing for the A4000 shows the dimensions of the 68040 coprocessor card supplied by Commodore. Note that these are not the maximum dimensions.

The form factor drawing for the A3000 shows the maximum dimensions for that system. The connector used for the local slot is a KEL 200-pin. Plug in cards use the male edge.

13.1 A4000 Local Bus Slot Form Factor (68040 Coprocessor Board)



13.2 A3000 Local Bus Slot Form Factor



CHAPTER 14

THE AMIGA VIDEO SLOT

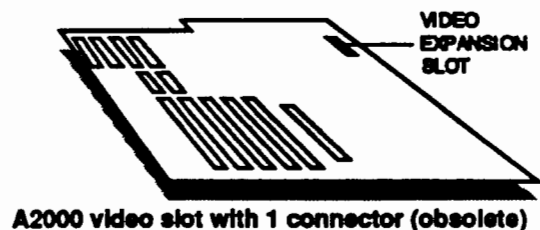
"I like to watch."

-Chauncey Gardener

This section details the signals found on the internal video slot of the Amiga 2000, 3000 and 4000 models. The video slot consists of two in-line, female edge connectors mechanically similar to the slot extension connectors of an IBM PC-AT.

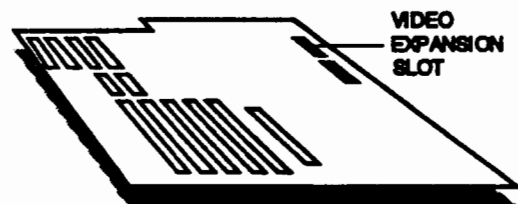
14.1 Evolution of the Amiga Video Slot

Originally the video slot was designed to provide the same functionality as the external 23-pin video connector but in a form that could internally house video boards such as modulators, genlocks and so on. This design required only a single 36-pin connector and the earliest models of the A2000 (4-layer board) do not have a second video connector in-line with the first. Very few of these early models were manufactured.



A2000 video slot with 1 connector (obsolete)

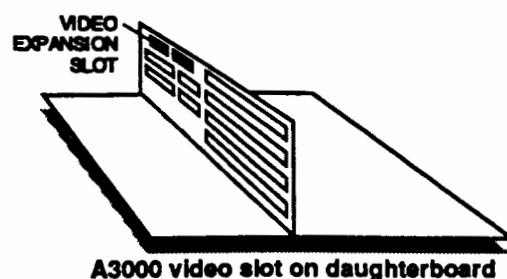
In later versions of the A2000, the video slot was expanded by adding a second 36-pin connector. A similar design was used in the A3000. So, in all A3000s and almost all A2000s, the video slot consists of two in-line, 36-pin female edge connectors. These contain all the signals available on the external video connector, plus all 12 bits of digital color and additional signals.



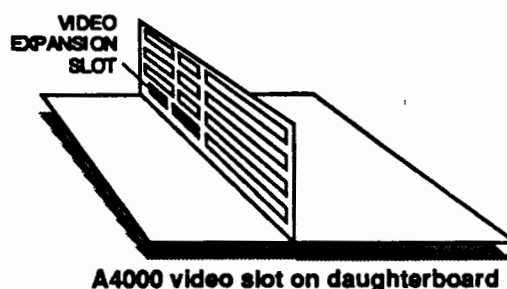
A2000 video slot with 2 connectors

In the A3000, the orientation of the slots was changed because of the addition of the expansion

daughterboard (see figure at right) but they slots work the same way as in the A2000. The arrangement of the expansion daughterboard used in the A3000 allows the video connectors to be in-line with a Zorro III expansion bus connector. Thus there is the potential for creating combination cards or more exotic video applications than previously possible in the Amiga 2000 system.



The video slot in the A4000 is very similar to the video slot in the A2000 and A3000 except that the second female edge connector has 54 pins instead of 36. The extra pins on the second connector brings out the additional 12 bits of RGB color available with the AA chip set. As with the A3000, an expansion daughterboard is used allowing the video connectors to be oriented so that they are in line with a Zorro III expansion bus connector. Also note that the video slot is positioned at the bottom of the daughterboard on the A4000 instead of the top as on the A3000.



It is possible to create a single plug-in-card that is compatible with all three Amiga models, but there are some mechanical complications. For one thing, the mounting bracket which attaches the video card to the opening at the back of the A4000 has been standardized (Bracket G44 Basic Blank, Globe Manufacturing). The old bracket used in the A2000 and the A3000 is not compatible (it is too large and bumps into the outer casework of the A4000).

The other problem is accessing all the new pins in the second video connector of the A4000. A video card with two sets of fingers designed for the A4000 will not fit into the smaller slots of the A2000 and A3000. The solution is to use three sets of fingers on the video card. That way the card will fit into the A2000 and A3000 while still having access to the new pins in the A4000. See the form factor drawings in chapter 2 of this section for details.

14.2 Video Slot Connector 1

Video slot connector #1 is the slot closest to the rear of the machine (see figures above). It is present on all models of the A2000, A3000 and A4000.

14.2.1 Power Connections

The video slot provides several different voltages for use by video cards. In all Amiga models, there is a single power supply for the main board and all other expansion ports as well as the video slot. The A2000 power supply is rated at 200 watts, the A3000 at 135 watts, the A3000T at 280 watts and the A4000 at 150 watts.

Video Ground

Video supply ground used by all video devices and the internal video circuitry. The Video and Digital grounds are common signals on all Amiga models except for a few very early A2000s (4-layer motherboard). This is available on pins 9, 12, 13, 17, 20, 21, 24 and 32.

Main Supply (+5V).

Main digital level power supply for the video slot. This can supply large currents, on the order of 2 Amps or so. The maximum supply current for the entire A4000 system is 20 Amps for all devices that use +5V including the motherboard. For the A3000, about 0.75 Amps is available for video with 17 Amps available for the whole system. For the A2000, about 2 amps is available for video. Pins 6 and 8 on the A2000 and A3000. Pins 6, 8 and 35 on the A4000.

Negative Supply (-5V).

Negative version of the main supply, for small current loads only; there's a total of 0.2 Amps for the entire A4000 system. (For the A3000, 0.2 Amps; for the A2000, 0.3 Amps.) Pin 31.

High Voltage Supply (+12V).

Higher voltage supply, intended for small loading only; there's total of 4 Amps (8 amp surge) for the entire A4000 system, much of which is normally devoted to floppy and hard disk drive motors. (For the A3000 3 Amps; for the A2000 8 Amps). Pin 10.

14.2.2 Clock Signals

These are various clock signals useful for synchronous timing of video peripherals.

/C1 Clock

For NTSC, this is a 3.58 MHz clock that is synced to the falling edge of the 7.16 MHz system clock. Also known as /CCK in some places. For PAL, these frequencies are 3.55 MHz and 7.09 MHz respectively. Pin 34.

/C4 Clock

For NTSC, this is a 3.58 MHz clock synced to the rising edge of the 7.16 MHz CDAC clock. For PAL, these frequencies are 3.55 MHz and 7.09 MHz respectively. Pin 19.

External Clock (XCLK,/XCLKEN)

The video slot provides for an external system clock, generally used to cause the entire Amiga system to become synchronized to something external. This should be something very close to the 28.64 MHz clock normally used to drive the system; the value used for XCLK can be a somewhat higher frequency, although anything too high will cause memory and other system timings to breakdown. XCLK will only be engaged as the system clock when /XCLKEN is asserted. XCLK is found on pin 33, /XCLKEN is on pin 16. There is no fixed phase relationship between XCLK and internal clocks and video outputs. Video interfaces must synchronize to the output clocks/video.

14.2.3 Video Signals

Access to the video signals are the main point of having the video slot. Most of these are also found on the 23-pin external video connector.

Analog Video

This is the analog RGB output, which consists of Red, Green and Blue signals, each of which generates a 0.7V p-p, 47 ohm terminated analog output. Found, respectively, on pins 7, 11 and 15.

Digital Video

These signals serve as digital output suitable for use with an IBM or Commodore 128-style, 4-bit digital (RGBI), color or monochrome monitor. Each of these outputs is 47 ohm terminated. The pin assignments are Digital Red on pin 29, Digital Green on pin 27, Digital Blue on pin 25, and Digital Intensity on pin 23. The digital Red, Green and Blue are made up of the most significant bits of the 12 or 24 bit video. Intensity is the "middle" bit of the Blue bits.

Blank (BLANK)

Specifies those times during the display when the video should be blanked. A4000 and AA machines only. Pin 26.

Separate Sync (/HSYNC,/VSYNC).

These are the separate, bidirectional, 47 ohm terminated video frame synchronization clocks. The horizontal sync, /HSYNC, is on pin 22; the vertical sync, /VSYNC, is on pin 26. As the names imply, these sync signals are active low.

Composite Sync (/CSYNC)

This is a digital signal that combines /HSYNC and /VSYNC. Pin 14.

Old Composite Sync (COMP SYNC)

Analog sync signal for the composite video. Not present on the A4000 and AA machines. Pin 28.

Burst

NTSC/PAL colorburst. To obtain the correct PAL colorburst signal, the video plug-in card must multiply this signal by 1.25 (i.e., $3.55 \times 1.25 = 4.433$ MHz). Pin 18.

Pixel Switch (/PIXELSW).

Background color indicator (color 0) on a pixel by pixel basis. 47 ohm terminated, /PIXELSW. Pin 30.

14.2.4 Audio Signals

Along with access to video signals, audio signals are also available on video slot #1.

Audio (LINELF, LINERT)

The audio signals are the Left and Right audio channels, on pins 3 and 4, respectively. This is the audio signal passed through the Amiga's low pass filter. (Raw audio is available on the second video connector, pins 34 and 36.)

14.2.5 New A4000 Signals (Formerly Reserved for Future Expansion)

The video slot implemented in the A2000 and A3000 has pins 1, 2, 5, 35 and 36 reserved for future expansion. In the A4000, all these reserved pins are in use as described below.

PSTROBE

Most of the signals of the parallel port (Centronics connector) are brought out on the second video connector, pins 23-30. The parallel port handshake line, PSTROBE, completes the set. This is available only on pin 36 of the first video connector of the A4000. In the A2000 and A3000, this pin is reserved.

Pixel Synchronous Clock (C280)

This signal on the A4000 provides a clock synchronized with the pixels coming out of Denise allowing digital video boards without the phase-lock loop circuitry required on earlier Amiga models. Pin 5 of the A4000 only.

RGB 16 and RGB 17.

The second video connector provides complete RGB signals on the A2000 and A3000 (4 bits each of R, G and B). But on the A4000, there are 24 bits of RGB color (8 bits each of R, G and B) so there are 12 new signals to bring out. Most of these new signals appear on pins 45-54 of the second video connector but RGB16 (Red bit 0) and RGB 17 (Red bit 1) appear on pins 1 and 2 respectively of the first video connector.

14.3 Video Slot Connector 2 (closest to the front of the machine)

The main purpose of the second video slot is to bring out all the RGB color information from the Denise custom chip. On the A2000 and A3000, there are a total of 12 bits of RGB color information (4 bits each of R, G and B). On the A4000 there are 24 bits of color information (8 bits each of R, G and B).

14.3.1 Additional Video Signals

Composite Video (non-AA)

This is the analog level, monochrome composite video signal also available on the Composite Video jack. Pin 13. Changed to SOG on AA machines.

Sync on Green (SOG)

Sync on green bit brogrammable via the Copper. A4000 and AA machines only. Pin 13.

Digital Video

Signals on this connector combine with signals on the first connector to provide 12 bits of RGB color information on the A2000 and A3000 or 24 bits of RGB color information on the A4000. The timing of digital video is not tightly specified on the A2000 and A3000. (On those systems, a phase-lock loop circuit is required to synchronize with the digital video signals.) On the A4000, pin 5 of the first video connector (C280) provides a pixel-synchronous clock for digital video boards.

For 12-bit RGB (non-AA), the Red, Green and Blue bits are R0-R3, G0-G3 and B0-B3, respectively. These bits appear on the first connector and the first 36 pins of the second connector. R3, G3 and B3 are the most significant bits of color information.

For 24-bit video (AA), the Red, Green and Blue bits are R0-R7, G0-G7 and B0-B7, respectively. The extra 4 bits per color provide greater color resolution. Therefore, the 4 least significant bits of each color are the new bits and appear on the new pins in the second video connector. Also some unused pins of the original connector space have some of these new bits. Note that the bits that were R0-R3, G0-G3 and B0-B3 on a 12-bit (non-AA) system are now called R4-R7, G4-G7 and B4-B7 on a 24-bit (AA) system.

Light Pen (/LPEN)

This is an input to the Agnus light pen input. This signal should go low in response to the lighting of a pixel on a video display monitor. The Agnus chip latches the raster position that was in effect when the /LPEN signal goes low, so an application can follow the position of a light pen on the screen. Pin 19.

14.3.2 Additional Audio Signals

Raw Audio (RAWLF, RAWRT)

These are the left and right audio channels before they're passed through the low pass filter on output. For many applications, the audio sampling rate is low, and as such requires a low pass filter to be in place at $f_c = 6$ kHz or so, to prevent audio aliasing. However, higher sampling rates are possible, and in such cases a much higher filtering frequency is required for best possible sound. The raw audio, left on pin 33 and right on pin 35, is buffered but unfiltered.

Filter Cutoff (/LED)

This is the /LED port line and also controls the two pole low pass filter on the standard audio channels. When asserted, the filter is in place; when negated the filter is bypassed. This is an input to the connector, useful to allow any Audio/Video card to monitor the audio filtering state. Pin 31.

14.3.3 Additional Grounding

Digital/Video Ground (GROUND).

These pins provide additional grounding for digital or video based devices. Pins 1, 5, 9, 12, 22, and 32.

Audio Ground (AGND)

These pins provide grounding in common with the separate onboard audio ground. Pins 34, 36.

14.3.4 Additional Clock Signals

These are various clock signals useful for synchronous timing of video peripherals.

CDAC Clock.

For NTSC, this is a 7.16 MHz clock that leads the 7.16 MHz system clock by about 70ns (90 degrees). Pin 15. For a PAL system, this is 7.09 MHz.

/C3 Clock

For NTSC, this is a 3.58 MHz clock that is synched to the rising edge of the 7.16 MHz system clock. Also known as /CCKQ in some places. For a PAL system, this is 3.55 MHz. Pin 17.

Timer Time Base (TBASE)

This is the real time clock time-base input, either 50Hz or 60Hz, depending on the country involved and the setting of the Time Base Jumper. The jumper can select either line frequency or vertical synchronization as the clock's time base. Pin 14.

14.3.5 Parallel Port Connections

Most of the signals from the bidirectional parallel port (Centronics connector) are available on the second video slot. (Note that PBUSY is on pin 18 of the first video connector in the A4000 only.)

8 Bit Parallel Port (PDO-PD7)

The 8 bit bidirectional parallel port most commonly used to drive a Centronics interface printer externally is accessible here. It can be used to control various aspects of a complex video interface device. The port lines PDO-PD7 are on pins 23 to 30.

Parallel Port Handshake (/ACK).

This is the acknowledge (/ACK) input, the same as the acknowledge input to the parallel port. Driving this with an output from a Video Card can cause a level 2 interrupt to occur through the 8520 CIA device this is connected to, based on the programming of an 8520 register. On pin 20.

Other Port Lines (BUSY, POUT, SEL).

Connector pins 18 (BUSY) and 16 (POUT) are general purpose I/O signals that together can also function as a synchronous serial data port driven by an 8520 CIA device. In normal printer use, the BUSY signal is used to indicate the printer paper is out. For serial port usage, BUSY is the serial clock, POUT is the serial data line. These should be driven with open collector devices if the Video Card uses them as inputs to the 8520. The SEL signal, on pin 21, is a general purpose I/O port usually used as a device select signal on the parallel port.

14.4 Video Slot Pinout

Video Connector 1 (closest to the rear of the machine)

Amiga 4000	Amiga 3000 and Amiga 2000	Early Amiga 2000 (obsolete)
1 R0	1 Reserved	1 Reserved
2 R1	2 Reserved	2 Reserved
3 Filtered Audio Left	3 Filtered Audio Left	3 Filtered Audio Left
4 Filtered Audio Right	4 Filtered Audio Right	4 Filtered Audio Left
5 C280	5 Reserved	5 Reserved
6 +5 VDC	6 +5 VDC	6 +5 VDC
7 Analog Red	7 Analog Red	7 Analog Red
8 +5 VDC	8 +5 VDC	8 +5 VDC
9 Video Ground	9 Video Ground	9 Video Ground
10 +12 VDC	10 +12 VDC	10 +12 VDC
11 Analog Green	11 Analog Green	11 Analog Green
12 Video Ground	12 Video Ground	12 Video Ground
13 Video Ground	13 Video Ground	13 Video Ground
14 /CSYNC	14 /CSYNC	14 /CSYNC
15 Analog Blue	15 Analog Blue	15 Analog Blue
16 /XCLKEN	16 /XCLKEN	16 /XCLKEN
17 Video Ground	17 Video Ground	17 Video Ground
18 BURST	18 BURST	18 BURST
19 /C4 Clock	19 /C4 Clock	19 /C4 Clock
20 Video Ground	20 Video Ground	20 Video Ground
21 Video Ground	21 Video Ground	21 Video Ground
22 /HSYNC (47 ohm)	22 /HSYNC (47 ohm)	22 /HSYNC (47 ohm)
23 B4 = DI (47 ohm)	23 B0 = DI (47 ohm)	23 B0 = DI (47 ohm)
24 Video Ground	24 Video Ground	24 Video Ground
25 B7 = DB (47 ohm)	25 B3 = DB (47 ohm)	25 B3 = DB (47 ohm)
26 /VSYNC (47 ohm)	26 /VSYNC (47 ohm)	26 /VSYNC (47 ohm)
27 G7 = DG (47 ohm)	27 G3 = DG (47 ohm)	27 G3 = DG (47 ohm)
28 BLANK	28 COMP SYNC(Analog)	28 COMP SYNC (Analog)
29 R7 = DR (47 ohm)	29 R3 = DR (47 ohm)	29 R3 = DR (47 ohm)
30 /PIXELSW (47 ohm)	30 /PIXELSW (47 ohm)	30 /PIXELSW (47 ohm)
31 -5 VDC	31 -5 VDC	31 -5 VDC
32 Video Ground	32 Video Ground	32 Video Ground
33 XCLK	33 XCLK	33 XCLK
34 /C1 Clock	34 /C1 Clock	34 /C1 Clock
35 +5 VDC	35 Reserved	35 Reserved
36 PSTROBE	36 PSTROBE	36 Reserved

Video Connector 2 (closest to the front of the machine)

1 Ground	1 Ground
2 R4	2 R0
3 R5	3 R1
4 R6	4 R2
5 Ground	5 Ground
6 G4	6 G0
7 G5	7 G1

Video Connector 2 (closest to the front of the machine)

Amiga 4000

8 G6
9 Ground
10 B5
11 B6
12 Ground
13 SOG
14 TBASE
15 CDAC Clock
16 POUT
17 /C3 Clock
18 BUSY
19 /LPEN
20 /ACK
21 SEL
22 Ground
23 PD0
24 PD1
25 PD2
26 PD3
27 PD4
28 PD5
29 PD6
30 PD7
31 /LED
32 Ground
33 Raw Audio Left
34 Audio Ground
35 Raw Audio Right
36 Audio Ground
37 Reserved
38 Reserved
39 Ground
40 Ground
41 Reserved
42 Reserved
43 Ground
44 Ground
45 R2
46 R3
47 G1
48 G2
49 G3
50 G4
51 B0
52 B1
53 B2
54 B3

Amiga 3000 and Amiga 2000

8 G2
9 Ground
10 B1
11 B2
12 Ground
13 Composite Video
14 TBASE
15 CDAC Clock
16 POUT
17 /C3 Clock
18 BUSY
19 /LPEN
20 /ACK
21 SEL
22 Ground
23 PD0
24 PD1
25 PD2
26 PD3
27 PD4
28 PD5
29 PD6
30 PD7
31 /LED
32 Ground
33 Raw Audio Left
34 Audio Ground
35 Raw Audio Right
36 Audio Ground

CHAPTER 15

GENLOCK INTERFACE GUIDELINES

"One man's 'magic' is another man's engineering."

-Robert Heinlein

This chapter describes the reset mechanism of the Video Beam Counters on the current version of the Agnus chip when an external device (i.e. genlock) applies sync pulses on the HSY* and VSY* pins. The Commodore-recommended method of interfacing genlock devices to the Amiga for current and future compatibility is described here.

15.1 Hardware Interpretation of HSY* and VSY* Reset Pulses

Agnus is configured to accept external syncs, by setting the ERSY bit in the BPLCON0 register. After setting the ERSY bit the horizontal and vertical counters will respond as described below.

Horizontal Counter, NTSC Mode. If the external HSY* reset is not applied at the end of a "short line", the next line will be a "long line", as normal. If no HSY* is applied at the end of a "long line", the horizontal counter will roll-over and is held reset at count 00. The counter will resume counting once the external HSY* pulse is applied.

If the external HSY* reset is applied at the end of a "short line", the following line is forced to be another "short line".

If the counter reaches the end of a "long line" when the external HSY* is asserted, the next line is a "short line" and will start at count 01 (as opposed to count 00). Otherwise, if no reset occurs the counter will roll-over to count 00 and stop.

Horizontal Counter, PAL Mode. In PAL mode the counter will operate with "short lines" at all times ("longlines" are disabled). If the external HSY* is not applied at the end of a line, the horizontal counter will roll-over to the beginning of the next line and is held at count 00, until the next HSY* pulse occurs.

If the external HSY* is applied when the counter reaches the end of a line, the next line will start at count 01, skipping count 00.

Vertical Counter, Interlaced Mode. In interlaced NTSC mode, if an external VSY* pulse is applied, the vertical counter will be reset to count 000 (first line) and the long-field condition is set at the beginning of the next horizontal line (at horizontal count 02).

In interlaced PAL mode, if an external VSY* pulse is applied the long-field condition is also set. Note that for both NTSC and PAL systems, if the external VSY* pulse is not asserted during the end of a short (or long) field, the vertical counter will roll-over and start a long (or short) field sequence.

Vertical Counter, Non-Interlaced Mode. In non-interlace mode the vertical counter is set to operate with either long or short fields, depending on the last value that the software wrote to the FRAME bit. In this mode, whenever an external VSY* pulse is applied, the counter is reset to line count 000 (first line) at the beginning of the next horizontal line.

15.2 Pulse Duration

The width of the active-low vertical (VSY*) pulse should be less than or equal to one line duration (63.556 uS NTSC, 63.999 uS PAL) but greater than one-half line duration (31.77 uS NTSC, 31.999 uS PAL) and should be asserted during the beginning of a horizontal line, at which time the vertical logic is triggered and the first line is started. The width of the active-low horizontal (HSY*) pulse should be greater than or equal to eight hi-res pixels (0.558 uS for NTSC or 0.563 uS for PAL) for proper operation.

15.3 NTSC and PAL Genlock Interface Guidelines

In order to synchronize the Amiga to an external source, the genlock device must provide reset pulses to the Amiga's sync lines which have the effects described in the section above. This section describes the proper method of applying these reset pulses.

For NTSC Amiga models, the genlock device must provide the Amiga with horizontal and vertical reset pulses with the following rates:

- ☐ HSY* line: Active-low pulse of proper duration every two lines (i.e. 127.11 uS or 7.8671 KHz).
- ☐ VSY* line: Active-low pulse of proper duration every two fields (i.e. 33.36 mS or 29.97 Hz).

Note that after the VSY* reset pulse the next field will be an even field (long field).

For PAL Amiga models, the genlock device must provide the Amiga with horizontal and vertical reset pulses with the following rates. Both reset pulses must be generated regardless of whether or not source video is being input to the genlock device!

- ☐ HSY* line: Active-low pulse of proper duration every line (i.e. 63.99 uS or 15.625 KHz).
- ☐ VSY* line: Active-low pulse of proper duration every two fields (i.e. 39.99 mS or 25.0 Hz).

Note that after the VSY* reset pulse the next field will be an even field (long field).

In addition to providing the Amiga with the proper horizontal and vertical reset pulses, the genlock device must also provide the Amiga with a system clock that is synchronized with the external video. One method of doing this would be to multiply the video source line duration up to the Amiga's system clock frequency, thus making a line-locked clock that will also be free-running at the correct frequency when no video is present. For correct operation of the Amiga as a multitasking system, the proper system clock frequency appearing on the XCLK pin must be the following for NTSC and PAL Amiga systems:

NTSC XCLK: 28.63636 Mhz
PAL XCLK: 28.375156 Mhz

Note that some PAL genlock designers provide the Amiga's XCLK input with a 28.250 Mhz clock frequency. This is not recommended for proper Amiga operation!

1

2

3

CHAPTER 16

VIDEO BOARD FORM FACTORS

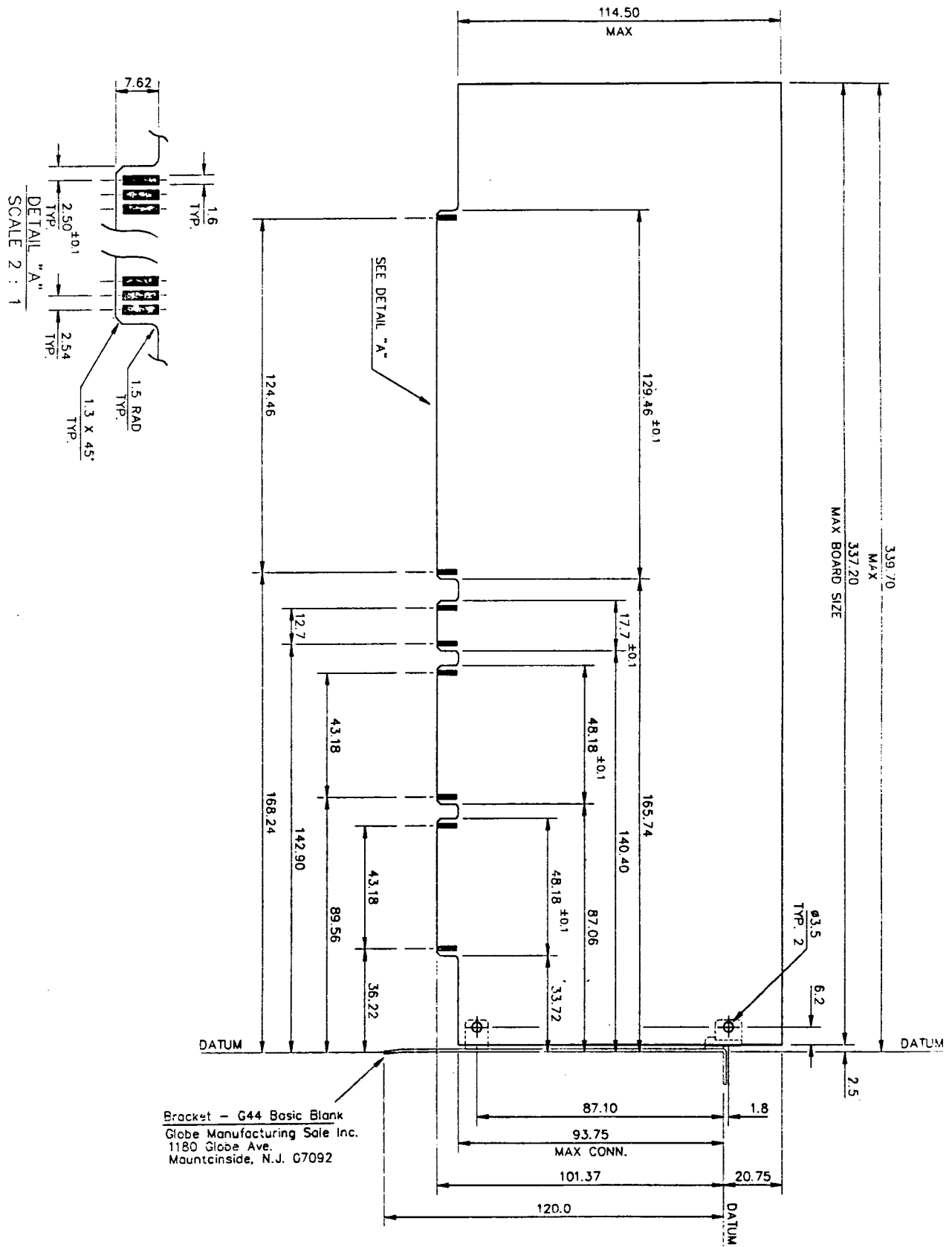
"The occupation of space is the real and final fact"

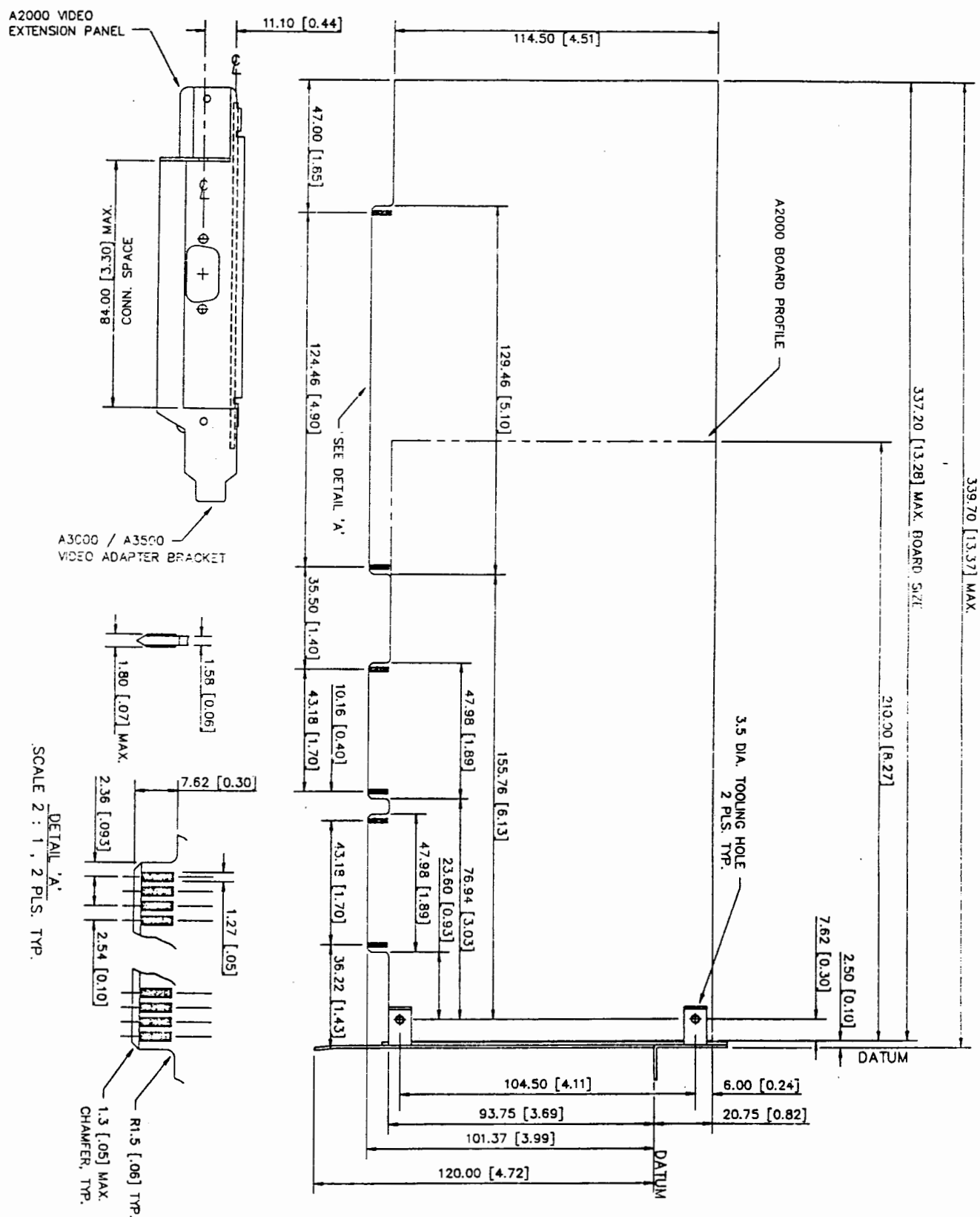
-W. H. Auden

The form factors for a video board for the A3000 and A4000 are shown on the next two pages. The profile for an A2000 video board is also shown in the form factor drawing for the A3000 for those hardware designers who want to create boards that are backward compatible with the A2000.

Keep in mind that there are differences in the way a video board's bracket attaches the board to the opening at the back of the Amiga. Because of these differences, brackets designed for the A2000 or A3000 may be too large to fit within the casework of the A4000. The A4000 video slot uses a standard bracket available from Globe Manufacturing (Bracket G44 Basic Blank, Globe Manufacturing, 1180 Globe Ave., Mountainside NJ 07092).

Amiga 4000 Video Board Form Factor





GLOSSARY

*"Thus the American, on his linguistic side, likes to make his
language as he goes along"*

-H. L. Mencken

G.1 Glossary

The reader may be unfamiliar with a number of terms used in this document. Every effort has been made to include all such terms here.

address	A byte-numbered memory location. The Zorro II bus is based on a 24 bit address, the Zorro III bus on a 32 bit address.
arbitration	The unambiguous selection of one request out of a number of possible simultaneous requests for a resource. There are two kinds of arbitration in a Zorro III system; bus arbitration and quick interrupt arbitration.
asserted	The active state of a state, regardless of its logic sense.
atomic cycle	A cycle or set of cycles that are uninterruptable, and thus treated as a unit; both Multiple Transfer and LOCKed cycles are considered atomic under the Zorro III bus.
AUTOCONFIG●	From "automatic configuration", the Zorro bus specification for how software and hardware cooperate to permit PIC addresses to be set by software and PIC type information to be determined by software. This is explained in Chapter 8, and in the <i>A500/A2000 Technical Reference Manual</i> , available from Commodore-Amiga.

backplane	The cage or motherboard subsection into which PICs are inserted. The Amiga 2000 and Amiga 3000 computers have integral backplanes, the Amiga 500 and Amiga 1000 computers require add-on backplane cages for Zorro II compatibility.
burst	A short name for Multiple Transfer Cycle mode. Essentially, within one full Zorro III cycle there can be any number of Multiple Transfer Cycles. Each full cycle has a complete 32 bit address supplied and a complete 32 bit datum transferred. Each burst cycle supplies only the 8 bit page address, but transfers a complete 32 bit datum faster than the standard full cycle would allow.
bus cycle	One complete bus transaction, indicated by the assertion of least one cycle strobe. For any single bus cycle, there is one address, one data value, one data direction, and one cycle type in effect.
bus hogging	When a bus master takes over the bus for an undue amount of time. The Zorro II bus leaves it completely up to the individual PIC to avoid bus hogging; the Zorro III bus schedules PICs with the bus controller to evenly distribute the bus load.
bus starvation	When a master can't get access to the bus, it is said to be starved. On the Zorro II bus, two busy masters can completely starve a third master. Complete starvation is impossible on the Zorro III bus, though a bus hogging Zorro II card can cause similar symptoms.
byte	A collection of eight signals into a logical group, and the smallest independently addressable quantity on the Zorro bus.
clock	A free running signal driven at a fixed frequency to the bus, used mainly for clocking state machines on Zorro II cards.
cycle strobe	A bus signal that defines the boundary of a bus cycle; the Zorro II and Zorro II modes on a Zorro III bus each have their own cycle strobes. The current bus master always asserts the cycle strobes.
data	The contents of a memory location. The main purpose of a bus cycle is to transfer data between two locations. The Zorro II bus is based on a 16 bit data path, the Zorro III bus is based on a 32 bit data path.
DMA	Direct Memory Access; devices that have direct access to Zorro III slaves are said to have DMA capability. These devices are also called masters.

DMA latency	This is the time between a bus request and a bus grant as seen by a PIC wishing to become bus master.
device	A PIC, e.g., a Zorro bus master or bus slave.
grant	The result of an arbitrated set of requests is a single grant; there are grants given for both the bus and quick interrupts.
Guinness	Attitude adjustment tonic, from Ireland. Said by some to be vital for sanity, if not normal human life.
hidden cycles	Cycles that occur on the local bus of a system, but can't be seen by devices on the expansion bus.
high	A signal driven to a logical +5V state is said to be high.
interrupt	An asynchronous line driven by a PIC to notify the CPU of some event, usually some hardware event governed by that PIC.
local bus	The main system bus of an Amiga computer is called the local bus. In general, the main CPU, video chips, chip memory, and any other built-in resources are on the local bus. The bus controller sits on both the local and expansion buses and manages the communications between them.
longword	Based on the Motorola conventions, a longword is equal to 4 bytes.
low	A signal driven to a logical +0V state is said to be low.
master	The device currently generating addresses for the expansion bus. There is only one master on the bus at a time, this being insured by the bus arbitration logic. The master also drives data on writes, the read, cycle, and data strobes, and several other signals.
motherboard	The main system circuit board for any Amiga computer. Resources on the local bus of a machine are often called motherboard resources.
negated	The inactive state of a signal, regardless of its logic sense.
nybble	A collection of four bits; one half of a byte. AUTOCONFIG® ROMs are physically nybble-wide.
paragraph	A sequence of closely related sentences, generally expressing and supporting one succinct idea. This term has no special computereese meaning in any rational, modern system.
PIC	Plug In Card. Any Amiga expansion card is called a PIC for short.

request	Asking for the use of some resource; the Zorro III bus has two kinds of requests, bus requests and quick interrupt requests.
slave	The device currently responding to the address on the expansion bus. There is only one slave on the bus at a time; an error is signalled by the bus collision detect logic if multiple slaves respond to the same address. The slave also drives data on reads, the transfer acknowledge strobe, and several other signals.
slot	A physical port on a Zorro backplane, which supplies independent /SLAVEN, /BRN, and /BGN lines, chained /CFGINN and /CFGOUTN lines, and is mechanically manifested as a 100 pin single-piece connector.
termination	Circuitry attached to a bus signal in order to minimize annoying analog things like ringing, reflections, crosstalk, and possibly random logic conditions which can arise when a bus is undriven.
timeout	A bus cycle terminated by the bus controller instead of by a responding slave device. If no slave responds to a bus cycle within a reasonable time period, the bus controller will terminate the cycle to prevent lockup of the system.
tri-state	A signal driven to a high impedance condition is said to be tri-stated.
word	Based on the Motorola conventions, a word is equal to 2 bytes.
Zorro	The name given to the Amiga bus specification. "Zorro I" refers to the original design for A1000 backplane boxes, "Zorro II" refers to the modification to this specification used for the A2000 and compatible backplanes, and "Zorro III" refers to the Zorro II compatible bus specification first used in the Amiga 3000 computer and later in the A4000.

1

2

3



Localizing Software for Japan

by Darren M. Greenwald

The follow information is confidential, proprietary, and subject to change. We are providing you this pre-release information so that you may register your interest with us, and start preparatory work now. Please fill out the registration form included with these notes if you would like to be contacted when we have final information.

Introduction

Commodore is actively working on a Japanese localized version of the 3.0 Operating System. Keeping in mind this is preliminary information, we plan on of having this version of the OS release ready by late spring of 1993, or early summer. In this session we will cover many of the technical issues that you will need to be aware of when localizing software for Japan.

The goal of the Japanese OS release is to provide a release that supports Japanese text output, text entry, and storage in third party applications, Intuition objects, and Shell windows. Whenever possible, compatibility and consistency with our existing 3.0 release of the Amiga OS will be maintained.

Some of you will be able to quickly port your software for use in Japan. If your application is already localized using locale.library, it is probable that non-text oriented applications can be used in Japan as-is, or with a little modification. Text oriented applications (e.g., a DTP program, or word processor) will require a significant rewrite.

Japanese Codesets

Locale.library has always supported the concept of associating a CODESET with a LOCALE, and with a CATALOG. Up till now, localized Amiga application have been able to ignore the codeset of a locale because all the languages we support are based on the 8-bit ECMA Latin1 characters.

The Japanese language makes use of more than 6000 characters including use of the ISO 7-bit coded characters, Hiragana, Katakana, symbols, and thousands of Kanji characters. Obviously the 8-bit ECMA Latin1 codeset is inadequate to store Japanese text.

We start by looking at how the problem of storing Japanese text has been solved on other personal computer systems. The Japanese computer industry has come up with a number of

solutions, all of which were designed with the intent of integrating Japanese text support in OS/application software that is fundamentally 8-bit oriented.

The first solution was the ANK 8-bit codeset. This solution replaced some of the extended graphics characters on MS-DOS machines with Katakana characters in the range of A1H-DFH. Of interest, use of the Latin1 characters is not possible with this codeset because glyphs in the range of 80H-A0H, and E0H-FFH are not defined.

Use of the ANK set by itself is not very useful, so a double-byte codeset known as JIS (which stands for "Japanese Industrial Standards") made it possible to store more than 6000 characters. JIS sequences are stored as byte pairs, however the byte values for these pairs collide with ASCII codes used in the West. To solve this, escape sequences are used around JIS streams.

Today a much more common codeset is known as SHIFT-JIS. SHIFT-JIS can be used without escape sequences because the first byte in a SHIFT-JIS byte-pair is in the range of 81H-9FH or E0H-EFH. This has the side-effect of making it possible to store the ASCII characters without ambiguity, and store the 8-bit Katakana character codes mixed with Shift-JIS. The downside of SHIFT-JIS is it conflicts with use of the C1 control codes (80H-9FH).

Finally there is a relatively new Japanese codeset known as EUC (for "Extended Unix Code"). EUC is a variation of SHIFT-JIS in which both bytes of a byte-pair must be in the range of A1H-FEH. EUC does not prevent use of the C1 control codes (such as 9BH used by console.device), and is more easily identified than Shift-JIS. It is somewhat non-standard, but is gaining in popularity.

Shift-JIS, and EUC are mixed single/double byte codesets. For both codesets, string parsing must be done with care. None of these codesets provides for use of the Latin1 Graphic Characters. This may seem to be a serious limitation, but just as you have no need for Japanese characters for normal use, the Japanese user does not need the Latin1 Graphic Characters. This of course does not mean that the Latin1 Graphic Characters would not be useful in a multi-language environment; it just means that multi-language support is outside of the scope of the Japanese codeset solutions mentioned above. If Latin1 characters are needed in the future, there is room in all of the double-byte codesets for new codes.

Unicode

The Unicode Standard is a relatively new 16-bit character encoding standard. In fact, the Second Volume of the Unicode standard was not published until 1992, and it is this volume

that completes the code assignments for the Chinese/Japanese/Korean characters. Unicode is *not* a multi-language environment standard. Text rendering, text entry, and language rules are outside the scope of the Unicode standard (beyond noting that software needs to be modified to support other languages).

Advantages of Unicode

- ☐ It is a published standard for storage of multi-language text.
- ☐ Unicode requires less storage space than the 32 bit ISO DIS (Draft International Standard) 10646.
- ☐ Provides a byte ordering mark (0xFEFF) to indicate byte order.
- ☐ Pure 16-bit encoding is simpler to use *in some cases* than the mixed single/double byte codesets used in Japan.

Disadvantages of Unicode

- ☐ Unicode is not a standard for storing text in Japan.
- ☐ Shift-JIS and EUC are mixed single/double byte codesets that often require less storage space than Unicode.
- ☐ There are no byte ordering issues when using Shift-JIS or EUC.
- ☐ Use of Shift-JIS or EUC is simpler to use *in most cases* than a pure 16-bit encoding model.

One problem you should be aware of is that the Unicode standard does not define a Unicode identifier sequence nor does it define a primary language sequence. Unicode streams can often be “sniffed out” if enough data is provided, though this does not solve the problem of needing to support JIS, Shift-JIS, and EUC text in our Japanese localization release. Japanese users will encounter all of these when importing/exporting data, so tools must be provided for data conversion regardless of the codeset we choose for Japanese text storage.

In some areas, the Unicode standard is not well defined. An example being string comparisons. String comparison can be complicated for some languages. It is even more complicated for strings that contain codes from more than one language, or codes that are used by more than one language.

Beyond string manipulation, a complete multi-language solution must address:

- ☐ Font support
- ☐ Adaptation of font rendering rules such as text direction, support for position dependent glyph forms, support for required styles, support for justification style, etc.

- ☐ Adaptation of text entry behavior on a per language basis.
- ☐ Support for modified keyboards if required.
- ☐ Language specific rules, such as word-demarcation, hyphenation if applicable, pattern matching, etc.

In the English language mapping of codes to glyphs is simple. There are some grammar specific issues such as capitalization of the first character in a sentence, but this is easily dealt with because separate ASCII codes are assigned for uppercase, and lowercase characters. On the other hand, the Arabic language has positional forms for a character for which Unicode codes are *not* assigned. Handling of glyph selection must be managed by an Arabic aware rendering process, not by modifying text storage.

Another example of language adaptation is word-demarcation for the purpose of selection, and word-wrap. In the Japanese language a word may be nothing more than a single Kanji character. The Japanese language does not make use of spaces, or other character separators in a sentence so a dictionary look-up algorithm is required.

What we conclude then is that use of Unicode does not simplify support for the Japanese language. It is unclear when, or even if Unicode will be adopted by the Japanese computer industry. It does have some potential advantages in a multi-language environment, however it also complicates porting of Japanese software. Therefore we plan on using the EUC codeset which will let us use single byte codes for the C0 Control group, ASCII characters, and C1 Control group. EUC is essentially pre-sorted, and easily converted to Shift-JIS or JIS which will be required in the Japanese Localization release.

It is interesting to note that Apple's *WorldScript*™ multi-language solution is not based on Unicode, but rather uses one of the Japanese mixed single/double codesets for Japanese text storage.

Locale Library and Codesets

Locale.library does provide support for defining the use of new codesets. The problem is that most Amiga applications do not check the codeset of a locale or catalog. The problem is further complicated because an application's built-in strings may not match the codeset of the catalog obtained with OpenCatalog().

To support Japanese, we will need to start making use of codeset identifiers other than 0. These are some problems you will need to be aware of:

- 1.) Many applications open the default locale, but do not check the codeset of the locale or catalog strings because up till now the only defined codeset has been 0.

Applications that use `locale.library` for static strings (e.g., menu text, gadget text, etc.) should have no problems. However, if you are manipulating the strings you get back in a catalog, checking the codeset of the locale and catalog may be required.

- 2.) It is possible for a language driver to be successfully started even though the catalog strings for that language have not been completely installed. The application may then fall back to its internal strings or `OpenCatalog()` may return strings for another language if the user selects alternatives. In either case, the built-in strings, or catalog strings actually obtained may not match the codeset of the locale. Even if you use `locale.library` functions (rather than `utility.library`), the result may be that you are passing strings to functions intended for use with another codeset or locale.

In part, this can be worked around in `locale.library` by modifying `locale.library` so that it will not return strings for a catalog that does not match the codeset of the language driver. What we cannot work around in the Operating System is the case of a language driver opening successfully, but `OpenCatalog()` failing because the catalogs are unavailable. You may need to check if the codeset of your built-in strings match the codeset of the locale.

- 3.) `Locale.library` does not address the issue of needing a font, and adapted rendering rules for languages other than our 8-bit codeset fonts.

In part, this will be worked around in `ASL`, and `diskfont.library`. A new tag will be added for the `ASL` font requester so that you can request `ASL` only show fonts for a specified codeset.

If you are using `Diskfont's AvailFonts()` function, use of the the `AFB_TAGGED` bit when calling `AvailFonts()` will be required so that you can ignore fonts that do not match the codeset you are interested in displaying.

- 4.) `Locale.library` provides only a subset of the functions needed for language independent manipulation of text. This is known, but will not be significantly changed for the Japanese OS release.

Japanese language support functions will be provided in the form of a shared library, and link library functions. We are looking at a more general language independent set of functions for the future.

Rendering Japanese Glyphs

A “glyph” is the visual representation of a character. For any codeset, the glyph used to represent a character may vary from font to font, or even from language to language. Just as different Amiga fonts provide different glyphs for the same character, some languages are even more complex in that glyph and glyph placement are dependent on position in a word/phrase.

Fortunately Japanese computer systems have adopted glyph rendering rules consistent with what the Amiga provides today. Use of left to right, top to bottom rendering is standard. Mapping of the EUC codeset to glyphs is unambiguous. Japanese fonts are most often non-proportional, and the need for alternate styles is mostly limited to use of underlined, and bold text.

Note that this is not entirely true. word processing applications may have a need to place outline generated Japanese glyphs more carefully than is possible using the graphics.library Text() function. Use of extruded, shadowed, outlined, etc., font styles are of interest in multimedia applications. Still, rendering of Japanese text is very similar to the features the Amiga provides today.

The first problem that will be addressed in the Japanese localized release of the Amiga OS is an extension of our existing bitmap font format. Our current font format does not allow for more than 256 glyphs per font. For text oriented applications, use of many Amiga fonts is certainly a possibility, but it is impractical for use in Intuition objects.

We are working on solving this problem by transparently providing support for fonts that render themselves. Such a font also provides vectors for font related functions such as the TextFit() function. From an application point of view, these fonts look just like any other font. They can be opened with OpenFont(), or OpenDiskFont(). They are used in the same way as any other font via SetFont(), and closed with the CloseFont() function.

Because applications “look in” TextFont structures, most of this structure will remain compatible. The TextFont structure looks like:

```
struct TextFont {  
    struct Message      tf_Message;  
    UWORD               tf_YSize;  
    UBYTE               tf_Style;  
    UBYTE               tf_Flags;  
    UWORD               tf_XSize;  
    UWORD               tf_Baseline;  
    UWORD               tf_BoldSmear;  
    UWORD               tf_Accessors;  
};
```

```

        UBYTE          tf_LoChar;
        UBYTE          tf_HiChar;
        APTR           tf_CharData;
        UWORD          tf_Modulo;
        APTR           tf_CharLoc;
        APTR           tf_CharSpace;
        APTR           tf_CharKern;
};

```

All structure elements up to, and including `tf_Accessors` will remain unchanged. For most applications this is compatible with existing practices. What has changed is that the glyph data may not be accessible by pointers in this structure, though at least one DUMMY glyph must be provided. For most applications this presents no problem, though font editors or applications that look at the actual bitmap data rather than using `Text()` will not work correctly.

Installation of these extended fonts will be handled by new functionality in `diskfont.library`. Function vectors will “hang off” the `TextFontExtension` structure created by the OS, so we do not expect any significant compatibility problems. Implementation details will be made available to interested developers when the design is finalized. In the interim, extended fonts will likely require:

- ☐ `Graphics.library` & `Diskfont.library` V40 (or greater)
- ☐ A standard Amiga bitmap font file, used as a handle
- ☐ A small extension file (that provides the name of font driver required to render the font, plus any custom information)
- ☐ A font driver in the form of an Amiga shared library
- ☐ Any additional private files required to render the font. The current plan is that no changes will be made to the existing `.font` file format, or font data files. In the case of the planned Japanese font driver, we will be using many small Amiga font files to store the Japanese glyphs, however this will be transparently handled for you, and the additional fonts will be invisible to the `AvailFonts()` function in `diskfont.library`. Font scaling, style specification, `diskfont` tags, etc., will all transparently behave just as they do now for our existing fonts.

A note on text styles. Algorithmic emboldening of Japanese bitmap fonts does not work well. Often the glyphs contain the bare minimum interstroke white space allowed for by the glyph matrix. Likewise underlining of Japanese fonts is not well defined because Japanese glyphs have no descenders.

Because of the large amount of RAM required to load the Japanese fonts into memory, these will need to be loaded from disk by a separate process managed by the font driver. For large Japanese fonts, or use of many Japanese fonts at the same time, this can mean slow rendering in a system with little free RAM. While loading, and expunging of font data will be handled for you, you will want to be aware of the need to leave as much free memory as possible for system use.

Various companies sell Japanese fonts in bitmap format, and outline data format. For the first Japanese release of the OS, we plan on shipping the system with at least two bitmap fonts, followed by a future product or release providing Japanese outline font support. Using the same font driver mechanism, on-the-fly Japanese outline font support can be implemented. Likewise additional bitmap fonts, and font families can be added later.

Note that this is pre-release information, and we have not finalized our decision of which fonts will ship with the system.

On other computer systems it is common to use Gothic, and Mincho Japanese fonts. Mincho fonts have smoother, rounded lines as if they were drawn with a brush. Gothic Japanese fonts have a sterile, functional look. For these typefaces, common sizes include Gothic 16x16 glyphs, and Mincho 24x24 glyphs. These are common for computer display use, though larger sizes are needed for DPI applications, quality printing, etc. One important point to note is that the glyph data for these sizes do not include white space! Most Amiga applications assume that the glyph data for a font includes at least one pixel of white space below, and around each glyph.

As many of you have already realized, there is no 8x8 Japanese font. This means some layout work is required for applications hard coded for Topaz 8; the worst case usually being application requesters. It turns out that even the 16x16 size is too small to represent the Kanji characters without some loss of data; 24x24 is better, and 32x32 is preferred. No wonder that Japanese computer users are interested in high resolution display devices; the language requires it.

The more observant may have noticed that the sizes listed above are even multiples of eight (8). Of course this makes sense for byte oriented computer systems, but there is a more interesting reason for choosing sizes that are an even number of pixels wide.

On character mapped display systems it is possible to take advantage of a display characteristic. If, for example, a character mapped display prints each byte as an 8 pixel wide glyph, then on a Japanese computer system we would have:

	0x41	0x42	0xA1 0xA2	0x43
	A	B	[EUC PAIR]	C
Pixels =	8	8	16	8 = 40 pixels wide.

As you can see, double byte Japanese character codes are printed as double-wide glyphs. It turns out that it has become common practice and because of this Japanese fonts include double-wide, double-byte versions of the ASCII alpha-numeric characters (and some symbols). Though system software generally does not interpret single byte, and double-byte alphanumeric characters as equal, double-wide alphanumeric characters are frequently used in word processing applications.

What all this means is that any localized programs you've written today can probably be localized for use in Japan without any significant change in the total number of bytes, or display width you have allocated for text. The Japanese language typically requires 10-25% more byte storage (per phrase or word) than English. Programs hardcoded for use of Topaz 8 can probably use the 16x16 font without requiring any change in display width.

It is assumed that the average Japanese display will be running at least a 640x400 display mode, or greater. We plan on shipping Japanese systems using at least this resolution, so you can assume that your application does not have to fit in a 640x200 display.

For the sake of maximum usefulness, Japanese fonts will be marked as monospaced fonts. For all practical purposes this is true, and allows use of Japanese fonts in a variety of applications that require monospaced fonts.

One of the problems we need to go back to is the need for white space around glyph data. Some other systems work around this problem by printing 16x16 Gothic Japanese glyphs in an 18x18 cell on the display. Likewise, 24x24 glyphs are often printed in a 26x26 cell. Because it is impractical to require all Amiga applications start providing white space independent of the font, we plan on providing algorithmic variations of Japanese bitmap fonts that transparently provide the white space (but share glyph data with the non white space versions of the fonts).

So in "FONTS:" we might have:

```
coral.font
coral_e.font
coral      (dir)
          16
coral_e    (dir)
          18
```

In this example, use of "coral_e/18" would print SINGLE byte codes 9 pixels wide, and DOUBLE byte codes would print 18 pixels wide. All glyphs in the font would be printed 18 pixels high. It is required that all glyphs in an Amiga fonts be the same height. Providing these as separate font families means that applications do not have to be modified to present the user with a choice.

You are encouraged to use the white space enhanced variations of these fonts for readability, however in some cases use of the non white space enhanced fonts is applicable:

- ☐ When 40 columns of Japanese text is required on a 640 wide display use of the 16x16 Gothic font is required.
- ☐ In some requesters and applications where readability does not significantly suffer.
- ☐ As a last choice when porting would be significantly complicated by using the wider fonts.

Text oriented applications may choose non-white space fonts and provide inter-character spacing through use of the RastPort TxSpacing field, and the graphics.library Move() function. Use of TxSpacing is an uncommon practice, but will be supported (details of inter-character spacing behavior when used with Japanese fonts remain undecided).

Another common practice is use of single "byte" output when calling the graphics.library Text() function (or other text related functions). This practice is strongly discouraged when printing Japanese text, however...

We want to ease porting time for existing applications, so we plan on providing some transparent support for single byte Text() output. The RastPort structure "dummy" byte has been reserved for this purpose. When Text() is called, ambiguous bytes will be stored here for subsequent calls to the Text() function. Each time Text() is called, the Japanese font driver will look here for a "pre-character" in the range of 0xA1 through 0xFE. If a pre-character byte is found, it will be prepended to the string being printed. A macro will be provided so that applications may clear, get, or set the pre-character value if needed (e.g., such as when Move() is called). As an alternative we are considering clearing the pre-character byte whenever Move() is called.

The problem is transparent support for single character cannot be implemented in a way that works well for all existing software. Many applications assume that cursors should be advanced, even though no character has been output. We have tried various combinations with different results in existing applications. The most versatile solution is to print ambiguous bytes as white space, and move backwards rendering over this space the next time Text() is called (however, this causes visual artifacts in some applications).

The best solution is to print entire strings of text, and provide entire strings of text when calling text fitting functions. Japanese programmers use a variety of standard functions to work around these problem on other computer systems, assuring unambiguous strings are printed. In general, you will have no problem using translated static strings in Intuition, or GadTools objects; these functions already print entire strings of text (or will do so as needed to support Japanese).

An interesting side-effect of EUC is that use of the right guillemets to indicate submenus is not suggested for use in Japan. GadTools will automatically provide an alternative character(s) for you when a Japanese font is used. You will probably want to search for other uses of Latin1 characters in your software such as your choice for a bullet character in password entry requesters.

It is understood that text oriented applications such as word processing software will require a significant rewrite for use in Japan. This leads us to the next topic, Japanese text entry.

Japanese Text Entry

The Japanese do not use 6000 character keyboards. Most personal computers in Japan use 101 key keyboards, or minor variations with new key caps. VACS Corporation is generally credited with being the founder of a solution known as the Japanese Front End Processor (FEP). A Front End Processor is a piece of software that lets the Japanese user enter words phonetically using Roman characters, Hiragana characters, or Katakana characters. The Front End Processor then converts this text to a list of likely Kanji choices during entry that the user can select from. Popular Front End Processors include:

Company	Product	Popular Platforms
VACS Corporation	VGE-Gamma	Many
ErgoSoft	EG Convert 4.0	Apple Macintosh
AI Soft	WX-II	Windows
Just	A-TOK	NEC-98

In addition to these, many other Japanese software companies offer their own FEPs including East's "EI", and Fuji Software's "Karen." Of the major FEP vendors, ErgoSoft's EG Convert 4.0 is very portable and offers many features. AI Soft's WX-II package offers emulation of A-TOK, and VGE, but is the least portable. VGE-Gamma is portable, well known, and offers a good balance of features and system requirements.

Front End Processor companies are focusing on making the entry of Kanji more efficient by providing new features such as auto-translation, improved look-up algorithms, editing features, etc. Beyond use of a Front End Processor, companies like Computer Intelligence Corporation (CIC) are working on handwriting recognition.

Some of the Front End Processor companies offer machine-independent versions of their code. While not reentrant, most of these products are potentially portable with the addition of machine specific keyboard handling, and display routines. All of these offer some form of string editing module that accepts virtual key codes in, and returns a modified string and attribute array for the purpose of redisplay. Others provide word processing support functions, and dictionary support functions.

When entering Japanese text, it is expected that the cursor moves whole characters; that the backspace/delete keys delete whole characters; that scrolling and end-of-line conditions be adjusted for Japanese text; that highlighting of text be Japanese character aware. You can see that many of the simple algorithms used in 8-bit codeset oriented text editors, and some word processors do not apply to Japan.

Compilers sold in Japan often come with a large set of Japanese string support functions (usually linked with the application). We are looking at providing similar functions, and codeset conversion functions.

A complete language independent editing solution is outside of the scope of what we plan to accomplish for a Japanese OS release by 1993. Such a solution either introduce significant OS incompatibilities, or require significant changes to existing Amiga application software.

At this point in time, we plan on providing the following:

- ☐ Access to the Front End Processor in the form of an Amiga shared library. This implies that the FEP can be upgraded, or even replaced with an improved product in the future.
- ☐ Transparent use of the Front End Processor in Intuition string gadgets when a Japanese font is used.
- ☐ Transparent use of the Front End Processor in CON: windows when a Japanese font is used. This implies transparent support in Shell windows, but not in all console.device windows.
- ☐ FEP input preferences, and FEP dictionary support tools.
- ☐ A commodity that will let the user enter Japanese text, and paste to the clipboard.
- ☐ We are researching integration of Japanese text entry in at least one of our system-provided text editors.

- ❑ **Modify console.device behaviors to support use of Japanese monospaced fonts.**

Operating System software will make use of a new function in graphics.library that will return the codeset associated with a font. When a Japanese font is used, high-level OS functions (including Intuition string gadgets, and CON: windows) will automatically switch to the Japanese text entry mode. It is appropriate to automatically switch to Japanese text entry mode when the codeset of the font is JAPANESE EUC. Just as you must have a Japanese font to enter Japanese text, you must have use of the FEP to enter all of the glyphs/character codes supported by the font.

This “switching” behavior implies that applications that are hardcoded to use specific fonts will continue to work as-is (at least in string gadgets, or CON: windows). Applications that use the Workbench Screen Font and/or System Default Font will often be able to take advantage of the Front End Processor for free.

You will want to be aware that console.device cannot provide applications with free use of the Front End Processor as console.device has no intrinsic awareness of line editing. Some changes are required in the console.device to support proper editing behaviors in console.device windows. An example would be use of a vertical bar cursor because block style cursors do not work well when editing Japanese text.

Changes in console.device behavior will be enabled when a Japanese font is in use. We plan on maintaining a high degree of console.device compatability when any 8-bit codeset font is used. A more complete list of console.device changes will be made available to interested developers when this information is finalized.

Use of CON: windows is highly recommended. CON: provides high-level black-boxed line-oriented text editing.

Use of Intuition string gadgets is also highly recommended, though use of string editing hooks may cause some compatability problems. Please feel free to discuss your string editing hook needs with us after the session, or add them to your list of comments when filling out the registration form.

In order to display Kanji choices, other computer systems are doing this by displaying these choices in a separate “box”. This allows the user to see his current string while making a choice. This implies that we may need to open a separate requester, or requester window(s) to retrofit Japanese text entry in string gadgets and CON: windows. For most software this can be done transparently, however if your application opens a custom screen, but is not layers friendly, please let us know.

Direct use of the Front End Processor will be of most interest to Japanese developers. Direct use of the FEP is required for word processing applications, DTP applications, and other text oriented software. In fact, we are told that word processing is the most common use of personal computers in Japan. This may be a good opportunity for some of you to pair your Amiga experience with a Japanese software company!

Gaiji

Because the total number of possible Kanji characters is so large (more than 50,000), it is not realistic to provide all of the Kanji characters in a font set, or even define codes for all of these characters in the JIS codeset. For everyday use, the 6400 some characters specified in the JIS codeset is adequate.

For the Japanese user who has need of Kanji characters not contained within the JIS set, the problem can be solved in one of two ways:

- ☐ Use a multi-font word processor, providing the additional glyphs in a separate font.
- ☐ Assign unused JIS codes to custom glyphs created with a font editor.

The second solution is known as GAIJI. By assigning custom glyphs to unused JIS codes, the new codes can be learned, and entered with the FEP. Application software does not have to be multi-font aware to print Japanese text containing gaiji, so long as the same font is used.

The second solution is common, but suffers from some obvious problems:

- ☐ There is no standard, so exporting/importing text data that uses gaiji is problematic.
- ☐ Bitmap drawn gaiji, and use of Japanese outline fonts do not mix well.
- ☐ The user may have to create many bitmap representations of the same character in a multi-font environment like the Amiga.
- ☐ Printing gaiji in text mode on Japanese printers that support a built-in set of Japanese glyphs is problematic.

Gaiji support is very important, though rarely used. The average user may create no more than a handful of glyphs, if any. Businesses may need at most a hundred or so extra characters, and so long as everyone in the company has the same environment, gaiji is useful. Even though we have technical objections with the use of gaiji, we acknowledge the need to provide system support for this feature.

Note that this problem is not entirely addressed by Unicode. The Unicode specification has reserved more than 20,000 codes for use by Chinese/Japanese/Korean countries, however it is not possible for Unicode to reserve approximately 50,000 codes for this purpose. Some 6000 private codes have been reserved in the Unicode specification, however use of private codes must be done by mutual agreement. There is no standard, so importing/exporting Japanese text that contains user defined codes is still problematic.

Japanese Outline Fonts

Japanese outline fonts are offered by various companies including Ryobi Imagix Company, Fuji Software Incorporated, Knox Computer Systems Incorporated, and Technonet. Outline fonts are available in various file formats including straight line formats, line/bezier curve based formats, and TrueType formats.

Japanese outline font data varies in size from as little as 1.2 Megabytes per typeface to over 7 Megabytes per typeface for the TrueType format fonts. Quality of the fonts varies. Some vendors offer outline data calculated from a 1000x1000 matrix, while other are focusing on desktop video, and use a smaller matrix of 256x256 or 64x64. Naturally outline font data based on a larger matrix is larger in size.

All the outline font vendors we have talked with tell us that their outline data does not include hinting, so hand-drawn bitmap fonts are preferable for typical screen resolution use. Use of outline fonts makes the most sense for printing, and desktop video. Because of the relatively high cost of these typefaces, they make the most sense as an aftermarket product. Japanese users can expect to pay \$150 or more per typeface.

Filesystems

It will be necessary for us to provide another Amiga FileSystem type that does not compare upper/lower case accented characters for equality. This is required so we can support EUC filenames. We do not plan on changing the name length limit of our FileSystem, or otherwise modify our FileSystem in any way that will affect your software.

We strongly recommend that you not use Latin1 characters in filenames when shipping products intended for use in Japan.

The more observant will note that if filenames are stored in Japanese, the Shell or requester displaying the filenames *must* be using one of the Japanese fonts. This is correct, however we do not intend to provide any Operating System kludge intended beyond the obvious:

Japanese users who use Japanese filenames will be running Japanese system fonts. This same problem is true on other systems. Japanese users have no need to store some files using Latin1 characters.

If you have an application that opens a custom screen but does not inherit the Workbench Screen font, you may need to revise your software if using the ASL file requester.

CrossDOS will need some modifications to support Japanese filenames, and read NEC PC-98 disks. Intel based systems are using Shift-JIS strings for file names, and internal storage.

EUC aware DOS pattern matching routines will be provided when the Japanese locale driver is initialized.

Printer Support

We are researching Japanese printers we will support with drivers. Primarily this means identifying Japanese printers that support TEXT mode printing in Japanese. For text mode printing, conversion from EUC to the printers internal codeset, or sequences can be managed inside of the driver.

Integration of gaiji, and text mode printing is still being explored. Integration of screen bitmap fonts, outline fonts, and internal printer outline fonts is being explored.

Other Issues

Various OS utilities and editors will require minor revision to support use of Japanese text. We have already converted most of the preference editors to use one of the Japanese bitmap fonts. We are researching support for Japanese text entry in at least one of our system text editors.

If you are interested in more information, a registration form is included below. Please fill it out including the name(s) of existing products you would like to port, or new products you would like to write. Please return your form to Darren M. Greenwald. These will be given to CATS. so we can contact you when we have final information.



JAPANESE LOCALIZATION

(Please fill out this form if you are interested in being contacted when we have final information.)

Company Name: _____

Contact Person: _____

Business Address: _____

Business Phone Number:() _____

Business FAX Number:() _____

Developer Number: _____ Business Hours: _____

Existing/New products you are interested in porting:

Additional Comments:

1

2

3



Amiga Application Distribution In Japan

by Jack Plimpton, President, Japan Entry

The Japanese fascination with videogames, home video, and full-featured consumer electronics gadgetry suggests glowing opportunities for the Amiga in Japan. Already, there is a small community of Amiga-lovers in Japan.

Commodore is planning to support this growing Japanese user community through the introduction of a fully Japanized operating system in mid-1993. Also, Commodore is exploring the creation of a strategic partnership with a major Japanese manufacturer for large-scale marketing, distribution and support of the Amiga in Japan. These developments promise significant opportunities for aggressive software developers to get in on the ground floor.

Apple Macintosh sales have recently skyrocketed to 250,000 units annually, one of the few bright spots in the PC industry. This has led Sharp, Toshiba and other leading manufacturers to license Apple technology. Amiga is now in the process of being similarly discovered because:

- 1) it is a true multimedia computer, and
- 2) it is far more price-competitive.

Technology-savvy Japanese opinion leaders already recognize the extraordinary suitability of the Amiga for "edutainment" and desktop presentation applications utilizing video and music.

In the home, products that leverage the Amiga's speed and WYSIWYG graphics, such as videogames and graphics tools, will have excellent sales potential. In video production, the Amiga's built-in NTSC support, audio/video synchronization, and CPU power make it perfect for video editing, subtitling and special effects. Already, the Video Toaster is selling well in Japan, and users are interested in additional products and alternatives.

Japanese Market Overview

The Japanese desktop computing hardware and software market is growing at over 20% annually. Already, the PC market is one-fifth and the UNIX workstation market one-third the size of the U.S. American software firms should achieve at least 10-15% of their revenues from Japan.

1990 Packaged Software Sales in Japan

Ranking*	Company (U.S. Supplier)	Sales
#1	ASCII (Informix)	\$79 mm
#4	Microsoft K.K.	\$72 mm
#6	Lotus Japan	\$64 mm
#7	AutoDesk Japan	\$42 mm
#11	SystemSoft (Clarix, etc.)	\$32 mm

*1990 Packaged software sales ranking in Japan according to Nikkei PC

Common Pitfalls of Doing Business in Japan

Japan is the world's second largest PC and workstation market and offers excellent opportunities for sales. Strategic and technical partnerships with Japanese firms are also possible, and can often provide necessary manufacturing know-how, as well as increased visibility and leverage for establishing new products in the marketplace.

Three potential pitfalls are encountered by a significant number of U.S. companies in dealing with Japan:

Pitfall #1: They came knocking on the door

U.S. companies often engage, on a non-exclusive basis, the first distributor who approaches them. They fail to map out a coherent long-term strategy in Japan, or to adequately assess the capabilities and the integrity of the Japanese firm's organization, and its customer and industry relationships. The U.S. company is being used to diversify or bolster a weak market position.

In Japan, once a partnership has been established, even on a non-exclusive basis, it stays in effect. Other distributors will not approach you, and the non-exclusive distributor achieves a de facto exclusive relationship -- without any volume commitments or guarantees to the U.S. firm.

It is essential that you get to know your potential partner: exchange visits, build a rapport, and understand how your products fit into their strategy.

Pitfall #2: Lack of qualified management

Many Japanese firms lack qualified middle-level management. The traditional lifetime employment and seniority systems make it difficult to recruit start-up personnel.

Look for the partner who has the right people in place from day one: strategic thinkers, sales people and technical staff.

Pitfall #3 Reverse engineering

The company most likely to adapt and sell your product in Japan is the one with the greatest technical skills, together with the appropriate sales channels and the desire to enter your market niche.

Any distribution contract must include non-competition clauses. These are difficult to enforce under the climate established by Japan's Fair Trade Commission.

Therefore, the best protection is to constantly stay ahead of the competition in price and performance in order to discourage your partner from going solo or switching to a competitor.

Localization

To effectively sell the Japanese, a U.S. software product must be localized. Localization includes:

- Double-byte support
- Japanese translation of manuals and help screens
- Portation to Amiga DOS-J platform
- Integration with Japanese input processor
- Japan-specific features such as clip art, word wrap, search and sort routines, etc.
- Feature additions such as support of popular Japanese peripherals and file formats, "doughnut" rather than pie charts, etc.

Japan Entry recently guided Authorware, a leading authoring tool vendor, to a relationship with ASCII, Japan's largest PC software and book publisher. ASCII (1) made a multimillion dollar investment in Authorware, (2) published dozens of CD-ROM titles, and (3) with Japan Entry's help convinced NEC, Japan's number one PC vendor, to bundle a cut-down version of its authoring tool on NEC's multimedia machine as a "2nd-generation Hypercard."

The cost of localizing

Localization costs can vary widely, from \$60,000 for a graphics-oriented product that merely requires translation to \$1 million for a complex product such as a desktop publishing application or character-based database that needs to be significantly modified.

A rule of thumb that can be used to estimate localization costs is \$12 - \$50 per page of documentation, and \$.50 - \$1.00 per line of code.

Localization options

The following are some of the widely-recognized options for localization of a software product:

In an exclusive distribution agreement, the *distributor* often performs the localization in order to maintain up-to-date versions of the software and perform bug fixes. A Japanese software vendor acting as a re-publisher often provides the best option with regard to quality.

The *U.S. company* should control localization when it needs to sign multiple non-exclusive distributors to achieve market coverage. The U. S. company will also find it advantageous to control localization when there is significant direct OEM business.

A third party *localization specialist*, under contract to the U.S. company, can be cost-effective, in addition to protecting source code from potential Japanese competitors.

For DataEase, a business software vendor, Japan Entry signed up Dynaware, Japan's leading GUI software company, the first Japanese software house to offer a bestseller in the U.S. which heretofore had only sold its own software.

Picking the Right Japanese Partner

Whether you choose to set up non-exclusive or exclusive distributors, or even a subsidiary, it is wise to have sales, technical and financial partners when tackling Japan. How should you evaluate a potential Japanese partner's marketing and technical expertise, its financial strength in supporting product launch activities, and its commitment? Here are the common partnership options for entering the Japanese marketplace:

Partner Option #1: Software/hardware vendor

This option provides the best mix of sales coverage and technical support. An exclusive localization and distribution contract with a software house or hardware vendor will provide access to multiple distribution channels and "free" localization expertise. The biggest risk in dealing with a Japanese vendor is that it may become a competitor if your product does not prove sufficiently profitable. It is necessary to respond to your distributor's requests for new features, local adaptations and price decreases.

Partner Option #2: A dealer, distributor, or trading company

We recommend that a company engage distributors and dealer chains on a non-exclusive basis. Distributors offer an existing sales channel and contacts, but lack overall market coverage. They also lack technical skills and long-term commitment. Nonetheless, a large trading company can be a powerful ally if it is willing to provide personnel and financing to establish a joint venture, thus guaranteeing long-term commitment and channel independence.

Partner Option #3: Large firm seeking diversification

Many "smokestack" firms such as Nippon Steel and Kawasaki Steel are diversifying into high-tech. These firms tend to have very deep pockets. However, it is best to leverage smokestack companies through investment or joint ventures, given their limited technical and marketing expertise.

Skill Checklist	Relationship	Sales	Technical	Financial	Commitment
#1 ISV/Manufacturer	Exclusive/OEM	***	***	*	*
#2 Distributor	Non-exclusive	***		***	
#3 Smokestack	Joint Venture			***	*
#4 Subsidiary	Wholly-owned				***

Partner Option #4: Joint venture or wholly-owned subsidiary

A dedicated Japanese operation enables you to maximize long-term profitability by eliminating the middleman. By establishing a subsidiary, you will have more control over marketing and sales efforts.

The downsides are the high cost and time-to-market issues resulting from the difficulty of finding personnel and facilities. Firms choosing this route should have well-developed sales channels.



1

2

3



Amiga User Interface Directions

by Martin Taillefer

The Amiga's user interface is the most palpable aspect of the system. Users view the Amiga through Intuition. It is therefore critical that Intuition, and its ancillary components, continue to grow and become more functional and easier to operate.

This document outlines the major enhancements planned to the user interface. We wish to show the general direction in which we are heading. There are no guarantees that any of the material presented below is actually going to appear in the operating system.

Goals

There are two primary goals to current user interface developments:

☐ Increased knowledge reuse

A key motivating factor behind the original design of graphical user-interfaces was to increase the consistency and integration of personal computing environments. The refinement of the UI must engender more consistent applications, with a higher level of integration.

☐ Shorter development cycles

Simplifying and abstracting the API can substantially reduce application development time and development costs. This in turn promotes more and higher quality applications to be created.

The focus of UI development is on high user-benefit items. What can we do in the operating system that directly improves the usefulness and attractiveness of the system to current and new users? What can we do to make the system look like the right answer to the user's problems? This is why much attention is devoted to small details that have often been overlooked in the OS developments.

For example, putting a stronger emphasis on the graphical nature of the operating system. Although this has little direct impact on the usability of the software, it substantially improves its apparent quality and polish, which is an important consideration in the marketplace.

Low-Level User Interface Components

Windows

Windows are the heart of the Amiga's user interface. Their appearance, responsiveness, behavior, and manageability, determine in large part how the system looks and feels to a user.

Appearance

Amiga windows are fairly elegant, although there are a few details that could improve the overall look of things:

❑ Variable Thickness Borders

Except for the top border, window borders are fixed in thickness. Depending on the aspect ratio of the current screen, this might be adequate, or might make the window look bad. The current thin borders are a problem in high resolution modes where they become hard to see. The border dimensions of windows should adapt to the aspect ratio of the current screen, and should vary in width based on resolution. This also includes making system gadgets wider or taller based on the resolution.

Another issue with window borders is the presence of the very wide and often useless border needed when a sizing gadget is in place in a window. This is a big waste of screen real-estate on low-resolution displays. It is also a factor of confusion as there is no visual distinction between the drag bar and the blank useless borders. The blank borders can be removed by using an alternate sizing method (see the next section).

❑ Better Looking Titles

Along with variably-sized borders should come the repositioning of the window title text. The text currently touches the top border of the window, which looks quite bad and has often been reported as a bug in the system.

❑ Consistent 3D Effect

The current rendering for window borders does not create a consistent 3D effect. There are some rendering bugs that defeat the 3D effect.

❑ Better Looking Window Movement

The look of the user interface would greatly improved if windows were to follow the mouse pointer as they are being moved, instead of simply an outline of the windows. Whether this is actually implemented depends a lot on the performance attainable.

❑ Better Looking System Requesters

A mechanism similar to some PD utilities (ARQ) can be added to the system, further enhancing the graphical appearance of the UI. DOS requesters can automatically start using standard glyphs in them to indicate “disk full”, or “please insert disk”, and the EasyRequest() function can be extended to accept glyphs.

Behavior And Manageability

Windows are central to the operation of the system. Their generality and functionality should be maximized in order to remove as many hurdles as possible from the user attempting to navigate through the system. The windowing system should be a transparent environment that doesn't call attention to itself. It should do its job, not impose limits to annoy the user, and generally stay out of the way to allow real work to be done.

A few critical improvements can be made to our windowing system to substantially increase the usability of the Amiga as a whole.

First and foremost is the restructuring of the depth arrangement strategy. The current scheme in place since Release 2 has serious flaws. The main problem came with trying to shoehorn window zooming into old applications. This required the replacement of the dual depth arrangement gadgets by a single multi-function depth gadget. It is now very hard to depth arrange windows in a predictable way. It often takes multiple clicks and a lot of trial and error in order to get windows in the order you want them to be. Basically, the windowing system gets in the way.

Most competing systems do not have explicit user control over window depth arrangement. The active window is always brought to the front. We believe that user control over depth arrangement is preferable, and required for compatibility.

Certain things can be done to help in depth arrangement:

❑ Revert To 1.x Depth Gadgets

The original dual depth gadget scheme should be restored. It provided for consistent and determinate behavior and was easily understood.

❑ Frontdrop Windows

Certain utilities, or components within larger applications, require that specific windows always remain in front of other windows. For example, a clock program might want to hover in front of all other windows. Or a tool palette inside of a multi-window paint program would want to stay in front

of all active canvases. Frontdrop windows would be another stratum of windows, along the same lines as current backdrop and regular windows. This stratum would be on top of regular windows. Although general prioritized depth families could solve this problem in a general manner, such a scheme is likely overkill.

An important feature is the ability to resize a window from any of its corners or sides. Resizing a window currently involves locating and revealing the sizing gadget. Most window borders are not used for anything except as delimiters. Overloading them with the ability to resize the window provides a very nice increase in flexibility.

Another feature is the support of windows extending beyond the limits of their parent screen. Given the fact many Amigas are used on low-res displays, it is quite important to allow windows to be dragged off the sides of the display. It is much more important to do this on a system with a small number of pixels than it is on a system with a megapixel display.

Off-screen windows are also central to supporting the new concept of window iconification. The Zoom gadget introduced in Release 2 is a user interface failure. It has unpredictable behavior. Beyond that, what it does is just not that useful. The ability to iconify a window however is much more desirable from a user-standpoint, and can be substituted for the current Zoom gadget's functionality.

Clicking the Zoom gadget of a window will cause the system to move that window to a totally off-screen position. An icon representing that application will be displayed on the same screen the window was on. When the user double-clicks on the icon, the system removes the icon and brings back the window to its original position.

New applications can ask to be notified of iconification events so they can free resources. For example, a paint program can free pens that it had allocated, or a serial program can free the serial port for use by other software.

Screens will also be iconifiable. Basically, every window on the screen will get iconified, and then the screen is closed and turned into an icon on the Workbench screen. There are compatibility issues such as applications caching screen pointers. If Intuition cannot be sure that the screen owner and all visitors are aware of this new feature, the screen will be delinked though the screen memory would not be freed.

Mouse Pointer

There are three main issues involving the mouse pointer.

The most important issue concerns the tendency for the mouse pointer to freeze during lengthy Intuition operations such as layer handling, or when Intuition is blocked waiting for a semaphore to unlock. This property gives a very bad feel to the system. Anytime a window is opened, moved, sized, or closed, the whole system apparently freezes for a few seconds. The mouse stops moving and just sits there. On displays with many bitplanes, many windows, and no fast ram, the wait can be well over one second, sometimes close to five or six seconds (an A1200 running an 8 bitplane DblINTSC Workbench shows this). With all that said, preliminary investigations show that correcting this problem can best be described as extremely difficult, if not impossible.

The second pointer issue involves the inclusion of standard mouse pointer imagery for common tasks. For example:

- ☐ Crosshair
- ☐ I-beam
- ☐ Object Selection Mode
- ☐ + symbol for range demarcation

The third issue involves pointer positioning from the keyboard. The current scheme is very difficult to use. A simpler and more functional approach is to use the left Amiga key qualifier and the numeric keypad keys for directional control of the pointer. Amiga-0 would be the equivalent of a left button click, and Amiga-Enter would be the equivalent of the right mouse button. Consideration in this scheme must also be given to adequately handle "sticky keys" (described later in the Preferences section). Specifically, it must be possible to extend the stickiness of the left Amiga key to cover multiple mouse moves in one press.

Gadgets

This section presents a series of new gadget classes aimed at improving the functionality of our low-level system controls. These classes are often reimplementations of existing code, but with an eye towards making them more extensible, flexible, enjoyable to use, and more enjoyable to look at.

GadTools Objects

Although GadTools provided the framework needed to support our UI look, its many inherent limitations often become problems that application writers have to work around:

- ☐ **No Relativity**

GadTools objects don't support any form of positioning relativity, making them difficult to use in sizable windows.

- ☐ **Not Extensible**

GadTools objects do not support inheritance, making them impossible to extend from within an application. You get what you get, nothing more.

- ☐ **Not Changeable**

Once a GadTools object is created, it is impossible to change many of its static attributes such as its label or font. The only way of accomplishing these tasks is to remove all gadgets from the window and recreate them from scratch.

- ☐ **Not Connectable**

GadTools objects do not support any form of object interconnections. This prevents the creation of "self-driven" user interfaces, where the application merely sets things up, and lets the UI objects do all the work by themselves.

BOOPSI objects have none of the problems denoted above. They are as easy to use and access as GadTools objects are, and quite a bit more flexible.

With this in mind, the obvious course to follow is to create a series of BOOPSI classes that parallel in functionality the existing GadTools objects. The new objects would not have the old problems and would add missing functionality along the way. Once the new classes are in place, GadTools can become one of their clients. The main reason behind this move would be to reduce the amount of redundant code in ROM. It also guarantees that the new classes provide all of GadTools' functionality, in a compatible and consistent manner.

The following sections go through the various new or modified BOOPSI classes, explaining their basic functionality. Many of these classes are direct supersets of GadTools gadget kinds, with extensions to address the needs of a more graphical and font-sensitive user interface.

One of the key enhancements that affects almost all gadget classes is the use of the label.image class for the rendering of labels and other components of various gadgets. The labelling class enables every gadget type to support multi-line labels with embedded graphics. This makes the system much easier to localize and make font-sensitive.

Another enhancement to all classes is support for font-sensitive layout, where the gadgets can be queried for various types of information about themselves, such as their preferred size, or minimum size.

Finally, system classes that are intended for subclassing, such as `gadgetclass`, will be rounded out to be more flexible and powerful. We intend on providing better documentation and clearer guidelines on subclassing system classes to create your own classes, while retaining future compatibility with super- class enhancements.

buttonclass

This class already exists and should be extended with:

☐ **New Look Borders**

The class needs to be able to automatically render a border around a gadget's hit box. To give a bit more finesse to the look of these relatively boring gadgets, rounded corners could replace the current square corners.

☐ **Variable Repeat Rate**

The class needs to adapt its repeat rate based on a user preference setting.

checkbox.gadget

Checkbox gadgets are fairly intuitive and easy to understand. A few issues need to be addressed:

☐ **Larger Clickable Region**

Checkboxes present fairly small targets to mousers. This becomes a severe problem on large displays with small rendering. It is also a problem for handicapped users. Release 3 provides scalable checkmarks which helps a lot. The obvious way to further extend the clickable region is to listen for clicks on the checkbox's label and treat them as clicks on the checkbox itself.

There are a few open issues with checkbox gadgets:

☐ How would multi-line text labels look next to checkboxes? Would they also require a box around them to demark them clearly?

☐ Since it is likely that glyphs and multi-line labels will require some form of boxing, then would it be wise to always put a box around checkbox labels? This could also be used to increase the clickable region of this gadget type.

☐ The current rendering for the unselected state of a checkbox is simply an empty box. Is this an adequate cue to the user?

radiobutton.gadget

Radio buttons provide an attractive alternative to cycle gadgets and listviews. They tend to consume more screen real estate and more keyboard shortcuts than cycle gadgets and listviews. Nevertheless, radio buttons should be used when possible because they are the most intuitive type of gadget for the job: they clearly indicate all possible choices, and their rendering increases the graphical content of a display. A few issues should be addressed to make this type of gadget more palatable:

- ☐ **Strumming**

The user should be able to strum the mouse across the items of a radio button group. This would increase consistency with other gadget types.

- ☐ **Larger Clickable Region**

Radio buttons exhibit the same problem as checkboxes, they also present fairly small targets to mousers. As with checkboxes, the real solution is to listen for clicks on the button's label and treat them as clicks on the button itself.

- ☐ **Delimitation**

Also like checkboxes, radio buttons are currently fairly unbounded. This becomes a problem when multiple columns of radio buttons are created. Different grouping schemes can be used to help make things clearer.

- ☐ **Disabling Individual Buttons**

Individual buttons of a radio button group can be disabled and enabled independently.

popup.gadget

Cycle gadgets offer a nice compact alternative to radio buttons. However, the fact that not all available choices are visible at once is nonintuitive. The solution to this problem is the replacement of cycle gadgets with popup gadgets.

Clicking and holding down a popup gadget brings up a popup menu. You can then move the mouse within the menu. Releasing the left mouse button while on top of an item makes that item the current value of the popup gadget. Clicking the right mouse button while the menu is up cancels the gadget operation and closes the menu.

A point of contention with popup gadgets is whether they should retain the cycling nature of their ancestor the cycle gadget. A possibility is to keep the cycling behavior if the user clicks in a specific area of the gadget and bring up a pop up menu if the user clicks elsewhere in the gadget. Experiments will be conducted to find out whether this behavior is useful, tolerable, and desirable.

listview.gadget

Listviews offer a flexible concept for presenting variable lists of items. GadTools listviews however, have always been limited. This restricts their use, and prevents the easy realization of many viable user interfaces. Some features can greatly enhance their usefulness:

- ☐ **Multi-Selection**

Multi-selection is the biggest feature missing in current listviews. This ability is needed, for example, in the ASL file requester. ASL must currently implement its own version of listviews because of this.

- ☐ **Drag Selection**

Along with multi-selection comes drag selection. A group of items can be selected by dragging the mouse over these items while holding down the left mouse button.

- ☐ **Automatic Double-Click Detection**

Many listviews benefit from directly supporting double-click selection of items. Programmers forget this useful feature. Providing direct support for the detection of double-clicks would encourage use of this. Double-click support is useful as a shortcut for experienced users, making the system more "upwardly mobile".

- ☐ **Horizontal Lists**

It should be possible to create listviews that scroll data horizontally. This helps in the creation of font-sensitive layouts.

- ☐ **Multi-Column Display**

Many lists contain multiple fields of information for each item in the list. Consider for example a file requester displaying a file's name, size, and creation date. Support for multi-column displays means that programmers suddenly have the convenience to create listviews with multiple columns of information, without concern for proportional fonts or inter-column clipping. It also means that the user can optionally be given control over the size and even existence of any of the columns of information. By dragging a separator bar, the user can easily allocate more space for filenames, and less for file dates. This is also a major issue in the ASL file requester.

- ☐ **Secondary Scroll Bars**

In better support of large fonts, it is necessary to support scrollers to let the user move from left to right in a vertical scrolling list, or up and down in a horizontal one. This lets a user push some columns of information outside the visible area of a list, but still able to access them via the scroller.

scroller.gadget

Scroller gadgets are seldom used by themselves, and are mostly used as components in listviews. A few improvements help make them more useful:

☐ **Border Support**

Scroller gadgets work in window borders and are able to be embedded in them. This is useful in many applications. It standardizes issues such as repeating speed of the scroll arrows.

☐ **Better Looking Arrows**

The arrows currently in use in scroller gadgets are not very attractive and stand out in a world of snazzy looking 3D gadgetry. Arrow imagery should be taken from sysiclass. sysiclass will need to acquire freely scalable arrows instead of the currently limited arrows.

☐ **Right Mouse Button Cancellation**

Clicking the right mouse button while one of these gadgets is activated terminates the activation and restores the scroller to its original value.

slider.gadget

The main improvements for slider gadgets will be:

☐ **Enhanced Imagery**

The imagery should be more evocative than a simple black box. This would improve the look of the UI. It would also serve to distinguish sliders from scrollers. These have a very different purpose in life, this fact is not clear to the user since both gadget types currently look so much alike. Options of the new imagery include tick marks, and track filling to the left of (or below) the knob.

☐ **Repeating Container**

Clicking and holding the mouse in the container area of a slider gadget should cause the slider to enter repeat mode and gradually move towards the mouse.

textentry.gadget

Text gadgets are used by most applications and are the most limited type of system gadget. Many enhancements are needed:

☐ **Multi_line Support**

The biggest limitation of current text gadgets is their inability to handle multi-line data.

This new class will support multi-line editing and will include the use of optional scrollers to enter large amounts of data. It will also offer automatic word-wrapping, which is an important feature to make multi-line editing easier.

❑ Text Editor Mode

The multi-line support enables the simplification of a common UI composite object. The listview style in Workbench's Information window is a fairly complex gadget. It has a listview, a text gadget underneath it, and an associated "New" and "Del" pair of buttons. Operation of this composite object is fairly complex and unnatural. To edit a string, you must click on it within the listview area, but type the text into the text gadget way at the bottom of the listview. Deleting items requires selecting an item, which puts it into the text gadget, and then clicking the Del gadget.

Instead of the above composite listview object, a multi-line text gadget in non word-wrap mode can be used. To edit a line, you click on it, and start typing right on the place where you clicked. Text is not word wrapped and causes the gadget to scroll towards the left as you add more text.

❑ Cut And Paste

Clipboard cut and paste will be done in the expected manner.

❑ Variable Cursor Styles

The cursor style will be under preference control. Vertical bar or block cursors, blinking or not.

❑ Standard Editing Rules

Text entry gadgets will offer a complete set of editing abilities, with clearly defined behavior. Applications such as word processors are encouraged to adopt the same style. This includes standard keyboard controls, standard word selection algorithm, scrolling behavior, etc.

❑ Templates

This feature will allow a template to be provided to the text gadget, which describes the format of the data the user is allowed to enter. This is the main use programmers have for the current text gadget edit hook, provided with a much simpler interface. A template is composed of commands and literal text. The commands represent fields where data can be entered, while the string literal are displayed as is by the text gadget. Templates would allow easy creation of phone number gadgets, or ZIP code gadgets, or floating-point gadgets.

❑ Deselection Detection

A bug that occurs in many applications is when the contents of a text gadget is only inspected whenever an IDCMP_GADGETUP event is received. Since it is possible for a user to modify the contents of a gadget without generating such a message, it is possible to have an application miss changes made to the text. IDCMP_GADGETOFF, discussed below, is the answer to this problem.

❑ Optional History

Text entry gadgets will be able to optionally remember their previous contents thus providing automatic history.

filereq.gadget

Over the years, there have been many requests for the ability to add an application's own custom gadgets to the standard file requester. This would not be a very flexible approach, limiting the possible growth of the file requester. A much better approach is to create a BOOPSI class which encapsulates the logic of the file requester.

The file requester class will allow the display of any given named directory. The class implements the file requester's scrolling list (or possibly two lists, one for directories and one for files), and implements the file requester's three text gadgets. A client of the class, such as ASL, would simply embed the object within its window and send it messages asking it to display various directories, show the parent directory, show the volume list, etc. The file requester object would communicate selections made by the user back to the client. The client will be responsible for the control buttons at the bottom of the current ASL file requester (OK/Parent/Volumes/Cancel).

fontreq.gadget

This gadget type will exist for the same reasons the file requester class will. The class will offer the font and size lists, and optional style, pens, and rendering mode controls. The ASL requester's sample section, as well as the control buttons, fall within the client's domain, and not in the class itself.

screenmodereq.gadget

Follows the same concept as filereq.gadget and fontreq.gadget. The screen mode-related controls are in the class, and control gadgets remain in the client. Due to the vastly increased number of modes available under RTG, a different approach will be used to allow mode selection. The current flat list of modes becomes quite difficult to use when the system can display hundreds of modes. The new approach will be criteria-based, where the user can ask

for a display that is 60Hz, 640x480 in 16 colors, and the system will figure out what is the best mode to display that.

colorselector.gadget

This class will provide the functionality to implement a standard color requester. It will be built-up from other object types including the colorwheel and the gradient slider.

printreq.gadget

This class will provide the functionality to implement a standard print requester within an application. It will allow per-printer options, and finally offer a consistent printer interface to the user. By providing printer-specific options, a printer driver could be written that controls a FAX. The printer-specific options would then allow control of items such as the destination phone number, header page, etc.

dragger.gadget

This class will allow the creation of draggable gadgets (icons). Draggable icons provide a very clear and elegant solution to many UI problems. They are unfortunately quite difficult to implement using the current system software, so almost no applications make use of them.

The dragger class will allow you to define an object on screen that the user can move around. Various features will allow control of the object behavior. For example, object motion can be restricted to within the current window, within windows of the same group, or allowed to go to windows of other applications.

calendar.gadget

This class will be in support of system tools such as Time prefs or Agenda. The class will provide a standard mechanism for the selection and display of dates. Features of this class will include:

- ☐ Renders a single page of a calendar
- ☐ User can click to select one or multiple days in the month
- ☐ Calendar pages can be made read-only
- ☐ Individual days can be ghosted or rendered in a different color
- ☐ Localization issues are handled transparently

pager.gadget

Some standard support needs to exist to create gadgets allowing the user to flip between various option pages from within a single window. "Option pages" means a display like PrinterPS which lets the user alternate between different sections of the available settings.

The technique currently in use in PrinterPS and Palette prefs involves a cycle gadget. This has the unfortunate effect that users think the current value of the cycle gadget is in fact an attribute maintained by the prefs editor, instead of a means to access additional settings. For example, the cycle gadget in Palette prefs contains two states: "4 Color Settings" and "Multicolor Settings". Users are under the impression that the cycle gadget determines whether the system is in "4 color mode" or "multicolor mode". What the gadget really means is that the 4 color settings or the multicolor settings are currently being displayed by the program. The setting of the gadget is not a preferences item, merely a control of the prefs editor.

To avoid the confusion, a different gadget style will be created to support this type of concept. The gadget will operate in a manner similar to tabs in a book. By clicking on the proper tab, you bring that page of the program on screen.

Font-Sensitive Layout

In order to completely support font-sensitive gadget layouts, the various system classes must be modified to understand a few new features. The main feature is one that asks each gadget what is its minimum size. This functionality is required in order to do true font-sensitive layout. Since only the gadgets themselves can know this information, they need to supply it to a client trying to use them in a UI.

Another benefit of this technique is that gadgets can easily get bigger when new functionality is needed. A gadget can grow and simply report its minimum size as larger. Using this method, gadgets can adapt to different screen resolutions and render differently (specifically, the thickness of the gadget borders can be variable to suit different aspect ratios)

Keyboard Control

Support for keyboard control of gadgets is currently very limited. Many things can be done by applications to fake it, some things are difficult, and some are impossible. Keyboard control is an important feature for power-users. Once again, this is a case of upward mobility of the system software. Keyboard control makes the system much faster to operate for experienced users, while not getting in the way of novice users.

We have already adopted the standard notation of an underlined letter to indicate the keyboard shortcut for a gadget. We are lacking adequate support to implement the standard's functionality correctly. Intuition doesn't have a very fine keyboard input focus. Our current interface has two possible targets to keyboard events: the active text gadget or the active window. This loose input focus can make certain things harder to do. For example, if there are two listviews in a window, which listview gets scrolled when the cursor keys are hit?

The UI style guide offers some guidance in the area of keyboard control, but there is a problem. A substantial amount of work is required on the part of application writers to implement proper keyboard control. And certain things are even impossible to implement legally (highlighting a button gadget programmatically). Current system support is limited to underlining a character within a gadget's label. That's not enough. We need to:

- ☐ Enable classes to inspect their labels to determine what is their keyboard shortcut. If no label is given, an explicit tag could be passed by the application giving the key to watch for.
- ☐ Enable classes to look at keyboard input as it is generated, and act in consequence. The key sequence would be swallowed and a message sent to the application in a manner similar to a direct gadget click.

With the above functionality in place, it becomes easy to set up keyboard shortcuts for an application's gadgets. Localization of these is also easy, as only the label of the gadget needs to change to have the keyboard shortcuts change with it. Input handling for the shortcuts is automatically done by the gadget classes, in the manner that best suits them. For example, a scroller would use its key shortcut to increase its value, while the same shortcut with the SHIFT key down would decrease it. The colorwheel class could use multiple qualifier to control the direction of its knob. Etc.

The style guide currently recommends using regular keystrokes as shortcuts for gadgets. This works best for windows that have no (or few) text entry areas. The left Amiga key can provide a stateless method of activating gadgets.

Images

The creation of sophisticated graphical and font-sensitive UI displays has always been a difficult task. Creating glyphs that are compact, quickly rendered, scalable, and correctly color mapped, is such a hassle that application developers generally don't bother.

Increasing the graphical appearance of the system can be made somewhat easier with the addition of a few basic BOOPSI image classes. The classes attempt to simplify the creation of font-sensitive displays containing graphics, as well as improve consistency of the UI.

label.image

A generic labelling class which can be used on its own, and is used by all gadgets classes for the rendering of their labels. The features of the labelling class will be:

- ❑ **Stand-Alone Or Linked Use**

Label images will be able to be used on their own in much the same way TEXT_KIND gadgets are today. They can also be used as labels to gadget types.

- ❑ **Multi-Line Labels**

Supports the creation of multi-line labels by embedding line-feeds within the label. Localization often generates much longer strings than the original English ones, and the ability to transparently split the text onto multiple lines greatly improves the appearance of both the original and localized version of an application.

- ❑ **Automatic Word-Wrapping**

Also in support of localization, automatic word-wrapping helps UI layout tremendously. The application provides a box and some text, and the class ensures that everything fits within it.

- ❑ **Glyph Layout**

It is important that any text label be able to incorporate glyphs. The mini layout engine needed to do the word-wrapping also handles embedded images automatically.

- ❑ **Keyboard Shortcut Indicator**

The class supports underlining a single character within a label, to denote a keyboard shortcut.

drawlist.image

This class allows the creation of simple graphics in a resolution independent manner. The client defines the imagery in terms of commands in an abstract coordinate system, and the class arranges to scale everything to the requested pixel size upon rendering.

glyph.image

This class provides a series of predefined system glyphs rendered using the drawlist class. These glyphs would be used throughout the system and would become quickly familiar to users. This further increases consistency of the UI. Standard glyphs could include:

- ☐ Warning symbol
- ☐ Fatal Error
- ☐ VCR control symbols (Play, Stop, Pause, etc.)
- ☐ Help symbol
- ☐ "For your information" symbol
- ☐ Key top symbol (Function key, Help key, Alt, Ctrl, etc.)

fuelgauge.image

This class will be used by any applications needing to show the progress of a task. The class will offer standard rendering for a fuel gauge, with the following features:

- ☐ Horizontal or vertical orientation
- ☐ Optional tick marks below or to the right the gauge
- ☐ Optional milestone indicators (0, 50, 100%) rendered below or to the right of the gauge.
- ☐ Optional current percentage done indicator rendered to the left or right of the gauge.

frameiclass

This class needs to learn about two new features:

- ☐ **Titled Frames**
The frames created by this class should be able to have a title embedded in the top line of the frame. This would support the rendering of radio button borders, and is generally useful to separate a window into gadget groups.
- ☐ **Rounded Corners**
Frames with rounded corners are needed to support the new look for button borders.

Menus

The Amiga benefits from a flexible menuing system which has always had the ability to remain hidden and out of the way of the user. This is of great value on low-resolution displays. However, menus do require some work to better support higher resolution displays and add general functionality.

One of the points to watch for in a user interface is the minimization of mouse travel to accomplish common tasks. The current menuing system can lead to very large amounts of mouse travel on large displays, since the menu strip always appears at the top of the display. In addition, the horizontal organization of the menu strip requires a full horizontal sweep of the display to access all menus.

Both problems can be solved by making menu headers appear in a stack under the mouse pointer. This avoids the need to move the mouse to the top of the display to view the menus, and significantly reduces required mouse travel to scan through all the menus.

Additional functionality becomes possible with this new menu organization:

❑ Moving Menus

While the user is holding down the mouse button over a menu page, if he moves the mouse over the “move menu” section of the menu, and presses the left mouse button, the menu starts to follow the mouse. When the left mouse button is released, the menu is dropped in place and menuing operations resume as normal.

❑ Nailing Menus

By moving the mouse over the “nail” icon in a menu and clicking the left mouse button, a menu can be turned into a window. This window is managed totally by the system. Menu items are converted into appropriate gadget types and selections made in this window are converted to corresponding menu selections and sent to the application. The ability to nail a menu to the screen enables the user to easily display often used commands on the screen where they become instantly available with a single mouse click.

❑ Amiga Menu

The main menu page for an application contains an “Amiga” menu, which provides a handy place for the system to add standard functionality to existing applications. Moving the mouse to the checkmark glyph reveals the Amiga menu. It contains standard commands including a list of common tools the user might want to start (Calculator, Agenda, etc.). This is similar to the current Tools menu in Workbench, but is much more general. The Amiga menu enables the easy addition of new items managed by the system. For example, a selection to bring up a list of running applications, or a list of available public screens, etc.

❑ Keyboard Navigation

Keyboard navigation of the menus is another feature which becomes easier with the new menu organization. A standard keyboard sequence is entered

which causes the main menu page to be displayed. The menu can be cancelled by pressing Esc. The selected item is moved around by using the cursor keys. Pressing RETURN either brings up a submenu, or selects an item.

☐ Visually Distinctive Mutually Exclusive Selections

There needs to be special rendering to identify mutually exclusive menu items. They currently share the imagery of checkable items, which is misleading to the user.

Another important feature to improve menus involves submenu delays. Intuition would interpret high-speed mouse travel as meaning “don’t change the state of which menu panels are up and which aren’t”. This would allow the user to “cut the corner” when heading towards a submenu, without fear of accidentally triggering some other submenu along the way.

Preferences

The ability to customize the work environment is one of the big features of modern user interfaces. It gives users an important sense of being in control. It allows them to become more productive as the computer can automatically adapt to their tastes, and not the opposite.

The Amiga has always had a rich variety of controllable attributes. In some respects, it has too many. One of the purposes of this section is to review the different preferences choices that we currently have, and see what additional functionality is needed. Another purpose is to find ways to increase the graphical contents of the preferences editors. Careful consideration is given to avoid overwhelming the user with too many options.

Fonts Prefs

Beyond an interface update, Font prefs needs to support the selection of more font types. The selection is currently very limited and doesn’t allow adequate selections to cater to high resolution displays in a pleasing manner. The new font selections include:

- ☐ Window Title Font
- ☐ Screen Title Font
- ☐ Menu Font
- ☐ UI Font
- ☐ Shell Font

Keyboard Prefs

This is a new program which provides half of the functionality previously contained in the Input prefs editor.

- ☐ **Integrated KeyShow**

This shows the layout of the different keymaps as they are being selected. It eliminates KeyShow as a stand-alone utility.

Mouse Prefs

This is a new program which provides the other half of the functionality previously contained in the Input prefs editor.

- ☐ **Swap Mouse Buttons**

Will let the user swap the functionality of the mouse buttons. This makes the system feel more natural to new left handed users. The selection button can then always be under the user's index finger, where it belong.

- ☐ **Sticky Keys**

Sticky keys is a feature which makes it possible for users with hand coordination problems, or users with one or no functioning hand, to use a keyboard. Sticky keys causes qualifier keys such as SHIFT or ALT to be typed in separately from regular keys. A user can then generate an "A" by pressing the SHIFT key, release it, and press the A key.

- ☐ **Cursor Selection**

This will allow a specific cursor style to be chosen for use in text gadgets, consoles. and applications. Choices include block and bar cursors, blinking or static, and the blink rate.

- ☐ **Mouse Blanking**

The MouseBlanker commodity program will be replaced by a checkbox option in Mouse prefs. When turned on, the mouse pointer is blanker whenever a key is entered. This will eliminate a Workbench program and will mainstream this useful functionality.

Palette Prefs

To complete the implementation of the V39 palette preferences scheme, more pens need to be added to the system:

- ☐ Text Gadget Colors
- ☐ Screen Title Bar Color
- ☐ Cursor Color

Locale Prefs

The individual parameters currently controlled by the country selection in Locale prefs will become individually controlled. This will let users pick exactly which date or number format to use, instead of forcing per-country selections. The current country selections will become presets that automatically adjust all the editable fields to match the exact specifications for each country.

WBPattern Prefs

WBPattern should support centering and tiling of its backdrop pictures. This would improve the appearance and usefulness of using smaller pictures. It would also look better when switching the screen mode of the Workbench screen, without changing the picture.

Sound Prefs

Multiple types of beeps should be supported, each indicating something different:

- ☐ Warning
- ☐ Fatal Error
- ☐ Attention
- ☐ Task Completion

The different sounds give important feedback to the user when something happens.

Window Prefs

A new preferences program offering the following options:

- ☐ **AutoPoint Integration**
AutoPoint is a feature many users like. The current implementation is less than ideal as it is external to the windowing system. It also consumes a fair amount of memory and CPU time. Integrating autopoint functionality directly in Intuition eliminates memory requirements, and reduces additional CPU time consumed to almost nothing. Putting the option in the prefs editor also eliminates an obscure program from the system disks.
- ☐ **ClickToFront Integration**
For much the same reasons as integrating AutoPoint in Intuition, it is wise to also integrate ClickToFront.

☐ **System Requester Positioning**

Many users want to have system requesters appear in locations other than in the top left of the display. The user could choose to have the requesters positioned in any of the screen corners, have them centered, or have them appear under the mouse.

☐ **Border Control**

Control can be provided to alter the size of window borders.

ScreenMode Prefs

The current method for specifying screen modes is not intuitive. The variety of modes is overwhelming, and the current organization doesn't help the user understand the selection he is about to make. The future holds many more modes in support of AAA graphics, so a new approach must be devised to simplify the selection of modes names. The technique used in the ScreenMode preferences can also be directly applied to the ASL ScreenMode requester which will benefit applications.

Monitor Prefs

Monitor prefs will be a replacement for Overscan prefs. It will be functionally the same, with a few changes.

The major change will be the ability to control the aspect ratio for the various scan rates. This could be done by adding an extra gadget to the main window of the form "Edit Aspect Ratio", which brings up a screen where the user could graphically view and edit the aspect ratio of the scan rate.

Printer, PrinterGfx, and PrinterPS Prefs

Printer and PrinterGfx would greatly benefit from a face lift to give them the same basic interface as PrinterPS has. The ability to graphically edit the print attributes makes print control substantially easier. The current method used is very terse, complicated, and almost impossible to fully understand without having a manual in arm's reach.

These three program will also be merged into a single entity, which will use the pager.gadget to flip between the different option pages.

Pointer Prefs

Pointer prefs will be extended to support editing of the various pointer types added to the system's pointer class.

Default Preferences

The default set of preferences shipped with the system will be examined and updated if appropriate. Of specific interest are things that make the system look better in its default configuration. Possibilities include:

- ☐ Use of a proportional font as default
- ☐ Use of a higher resolution mode as default
- ☐ Use of more colors by default
- ☐ Use of a default Workbench backdrop pattern
- ☐ Use of centered system requesters

Programmability

There are a large number of subtleties in the Amiga's API. A few things can be done to substantially reduce programmer exposure to these subtleties. Following is a brief summary of the miscellaneous enhancements planned that will simplify the lives of programmers.

Windows

☐ RastPort Clipping

Two new features of future versions of graphics.library are RastPort-based clipping and RastPort-specific rendering origins. Once combined, these two features offer a lot to the Intuition programmer. Functionality similar to GimmeZeroZero windows can be provided using these features, without suffering all the performance problems of traditional GZZ windows. It also becomes possible to create separate panes within a window, each pane providing its own clipping, and its own rendering origin.

☐ Smarter Window Refreshing

Simple refresh layers could be made smarter. They would work mostly like current smart refresh windows, but would have the ability to dispose of allocated off-screen buffers when a memory panic occurs in the system. The reason for this window type is to enhance the performance of the windowing system. When possible, the very fast smart refresh algorithm is used, but if a memory failure occurs, the window behaves like if it was old style simple refresh.

❑ **Window Limit Specifications**

The ability to specify that the minimum and maximum dimensions of a window are to be interpreted as meaning “dimensions of the inner window region” would be of great help. These are currently very difficult values to set up correctly.

❑ **Window Fonts**

The ability to specify fonts to use in the window’s title bar, and in the window’s interior would add useful functionality and avoid much programmer work.

❑ **Window Ports**

Large applications frequently take advantage of the ability to share a single UserPort among several windows, but this adds some arcane complexity and ugliness to the code. A WA_UserPort tag would clean this up nicely, as well as mark that window in need of CloseWindowSafely() style processing inside Intuition.

❑ **Window Menus**

The ability to specify a menu strip to use in a window when the window is opened and the ability to close a window without having to call ClearMenuStrip() are two simple enhancements to make life simpler.

❑ **Window Event Buffering**

There is a need to buffer events that occur in a window prior to the window being completely ready to accept input. For example, when users bring up the file requester via keyboard shortcut, they start typing in their filename immediately after hitting the shortcut key. The problem is that the file requester is not ready to accept input yet. The result being the first few characters entered by the user for the filename are lost.

It is also fairly tricky to activate a string gadget in a window that is opening. Although this is better in V39, problems still exist. The application has to wait to receive an “IDCMP_ACTIVEWINDOW” event before it should try to activate the gadget. This is too much voodoo.

A way to solve both problems is to introduce a new OpenWindow() tag that means “this is the gadget that should be activated”. Intuition would then buffer up any input events that occur while the window is in process of opening, and would send them all in one big gulp to the window when it is ready to receive them. It would also auto-activate the appropriate gadget, to make sure any keyboard sequences that are intended for it actually reach it.

❑ **HideWindow()/ExposeWindow() Functions**

These functions would move the given window totally off-screen, and would bring it back to its original location. The `ExposeWindow()` function can be used in concert with the ability to open a window in the hidden state. This lets an application completely render its window imagery, and once done rendered, blast it onto the screen in one blit by using `ExposeWindow()`. Not only will rendering in the window go faster because of the reduced number of clipping rectangles, it will also look a lot nicer to the user.

❑ **ToolBox Windows**

ToolBox windows provide services to other windows. For example, a strip of tools shared across many document windows in an application. Intuition can help coordinate the toolbox window with respect to its parent.

❑ **Unified Windows and Requesters**

This concept means having a window with the basic desirable properties currently only available through a requester: locking of parent window (either as a new property of some windows or through a `LockWindow()` call), and some kind of parent/child arrangement for depth/size.

❑ **Better support for autoscroll screens**

We can think of a few ways to make the use of autoscroll screens more convenient, including autoscrolling before the mouse has hit the very edge of the display, and automatic scrolling of the screen to bring newly opened windows into view.

❑ **Screen locking**

This is basically an Intuition-friendly `LockLayers()` function. The advantage over `LockLayers()` is that it would be safe to continue to transact with Intuition without fear of deadlocking. This function would be used by programs which need to do trans-window rendering such as custom pop-up menus or draggable icons.

Gadgets and Images

❑ **IDCMP_GADGETOFF**

A partner of `IDCMP_GADGETUP`. It tells you when a gadget's activation state is terminated, but no `IDCMP_GADGETUP` was sent.

❑ **DrawImageStateFrame()**

New function that sends the `IM_DRAWFRAME` method to the image object, instead of `IM_DRAW`.

❑ **IDCMP_MOUSEMOVE**

IDCMP_MOUSEMOVE events generated because of a gadget should have the gadget pointer in the IAddress of the IntuiMessage, if the window requests it.

❑ **BOOPSI**

Autoknob Support Allow the combination of the sizing properties of an autoknob with the custom image capabilities of a BOOPSI image. Currently, you can have either one but not both.

❑ **GMORE_FULLYRENDER**

Gadget flag which says "I fully render in my hitbox" or "I fully render in my bounding box." We could skip erasing GREL gadgets in the area that intersects their new position (or the new position of any other gadget with this property).

❑ **SGH_INITIAL**

String edit hooks should receive a SGH_INITIAL message to validate contents on startup, and possibly a SGH_FINAL on exit.

Tool Port

A new unifying concept of IPC is being developed which will greatly simplify the job of writing applications on the Amiga.

A tool port provides a central communication point for an application. All IPC directed to this application are funneled through the port. The definition of the port interface allows unlimited growth and expansion. Routers can easily be added by the application to direct different message types to different locations. All system communications, such as IDCMP or ARexx messages, will come into the tool port in a standard format.

The tool port will finally give the system something it always lacked which is a unique handle by which an application formally identifies itself, and makes itself available to the outside world. The tool port will provide a set of pre-defined events, such as "shutdown", "show yourself", "open a file", etc., enabling the creation of system control tools.

The tool port will also simplify the programming model. All input coming into an application will come from a unique source. This serialization of messages guarantees correct processing sequence across all applications.

General Changes

There are a few general improvements that will affect most tools that come with the system software, and will benefit end-users and application writers.

Font-Sensitivity

All GUI-based system tools will become font-sensitive. This will let them adapt to any font the user chooses for his system. The layout task will be off-loaded to a new library called `layout.library`. This library will offer a generic rectangle layout engine, enabling effective font-sensitive layout.

`layout.library` will operate on a hierarchical organization of rectangles. Each rectangle describes its place within its parent rectangle. A rectangle can define its size in relation to its siblings, relative to the parent size, with a fixed number of pixels, etc. The layout procedure involves an iterative negotiation between the library and the rectangles, that causes the size and position of the rectangle to constantly adjust, until the requirements of all the rectangles are met.

On-Line Help

V39 added the necessary hooks to implement full context-sensitive help in applications. We plan on making all system tools provide help by making use of these mechanism. Some minimal additional support will likely be added to the new BOOPSI classes to make them easier to deal with. For example, each class can provide help about itself. In addition, some minor extensions to AmigaGuide are likely to happen, in order to make on-line help more usable. For example, support for simple help windows, where there are no prop gadgets, no system gadgets, and no AmigaGuide gadgets, could be handy to create light-weight help boxes.

Workbench

We are planning a complete rewrite and substantial redesign of the Workbench interface for the future. We are expecting a increase in performance, reduction in quirkiness, and substantial increase in functionality. However, plans are currently not detailed enough to be discussed.

As part of the Workbench rewrite, a tie-in with the ASL file requester will likely occur. The file requester will become a client of Workbench and will use its code to display file information. This will offer the user a consistent visual interface to the file system.

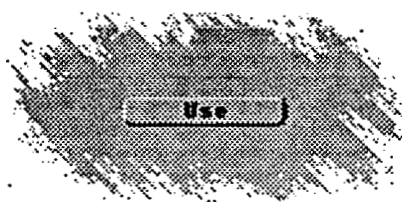


1

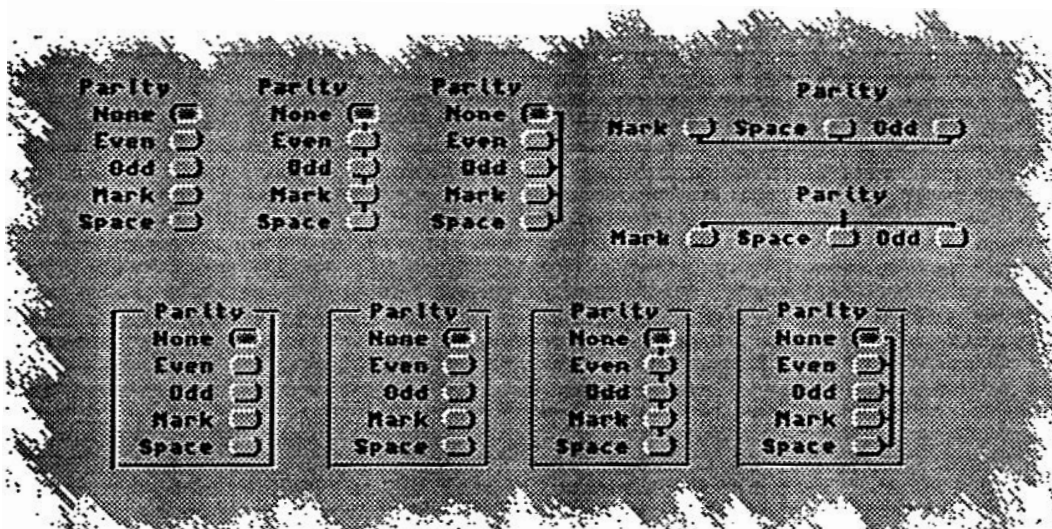
2

3

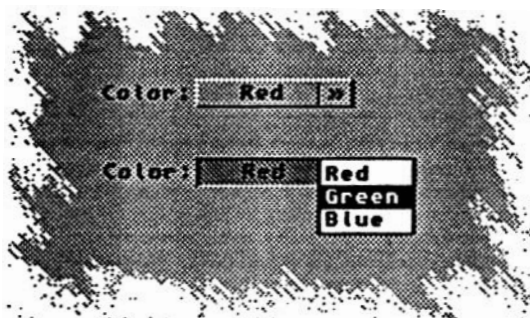
Amiga User Interface Directions Addendum



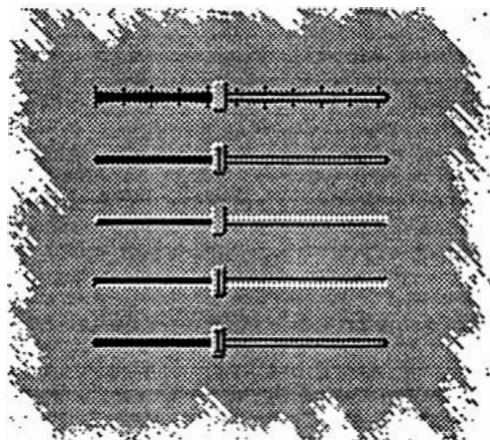
Page 7: buttonclass gadget with new look border



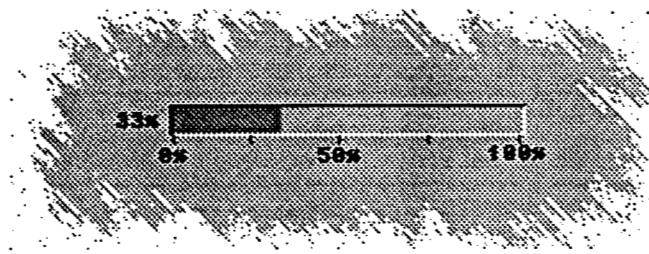
Page 8: radiobutton.gadgets in different grouping schemes



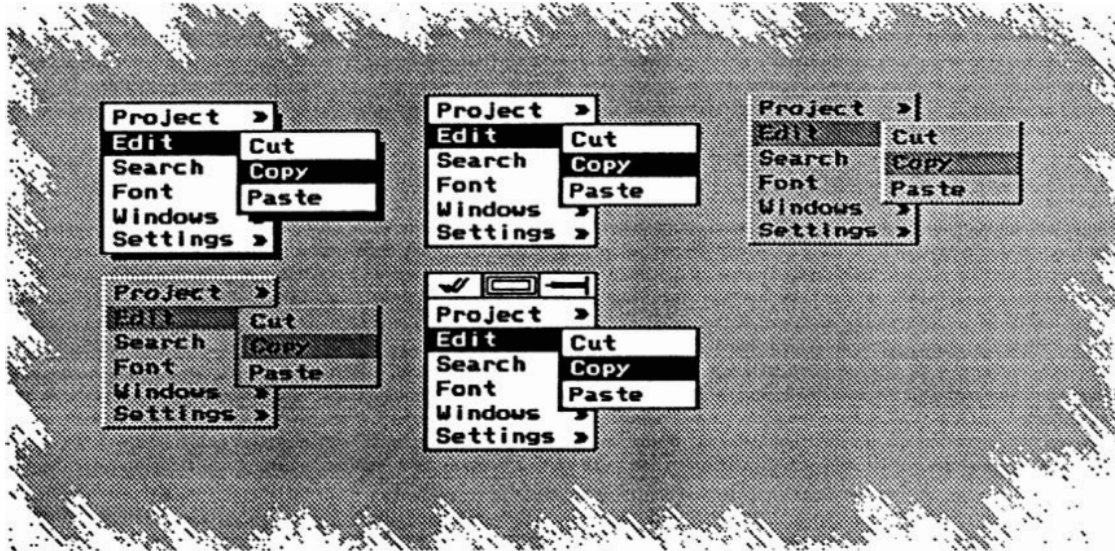
Page 8: popup.gadget



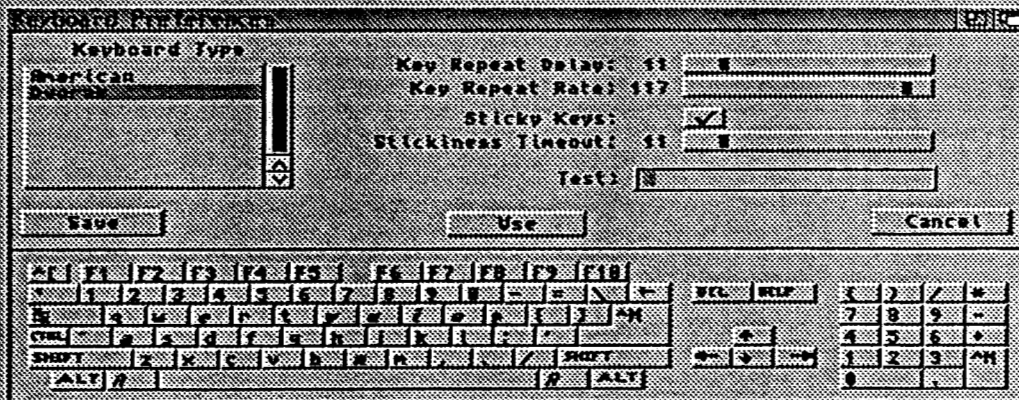
Page 10: slider.gadgets with enhanced imagery



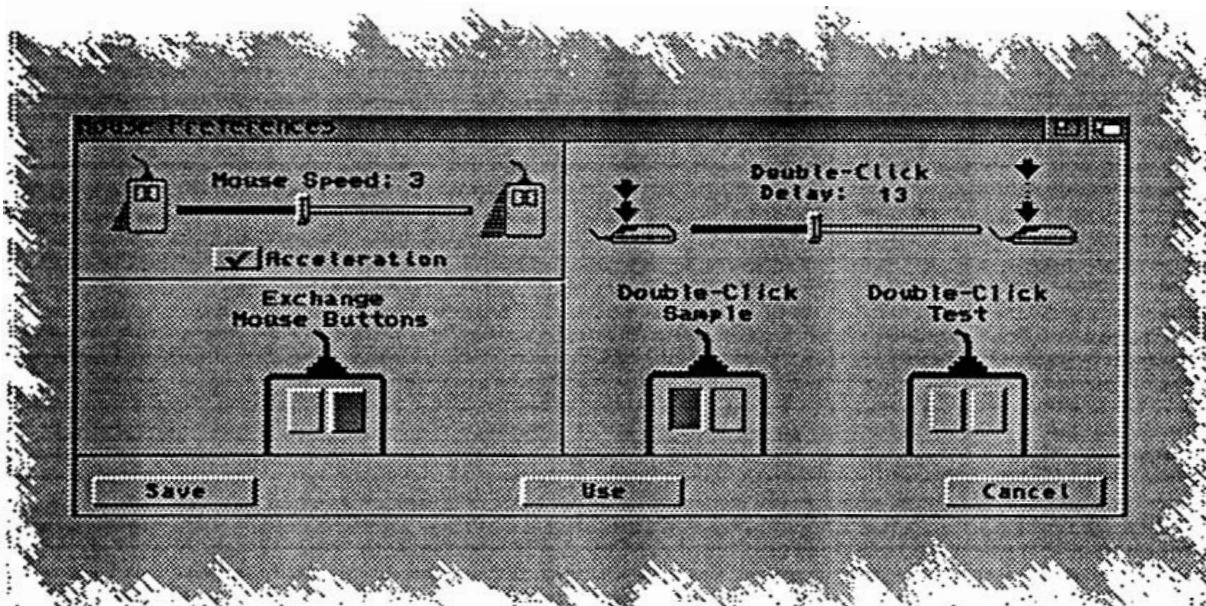
Page 17: fuelgauge.image



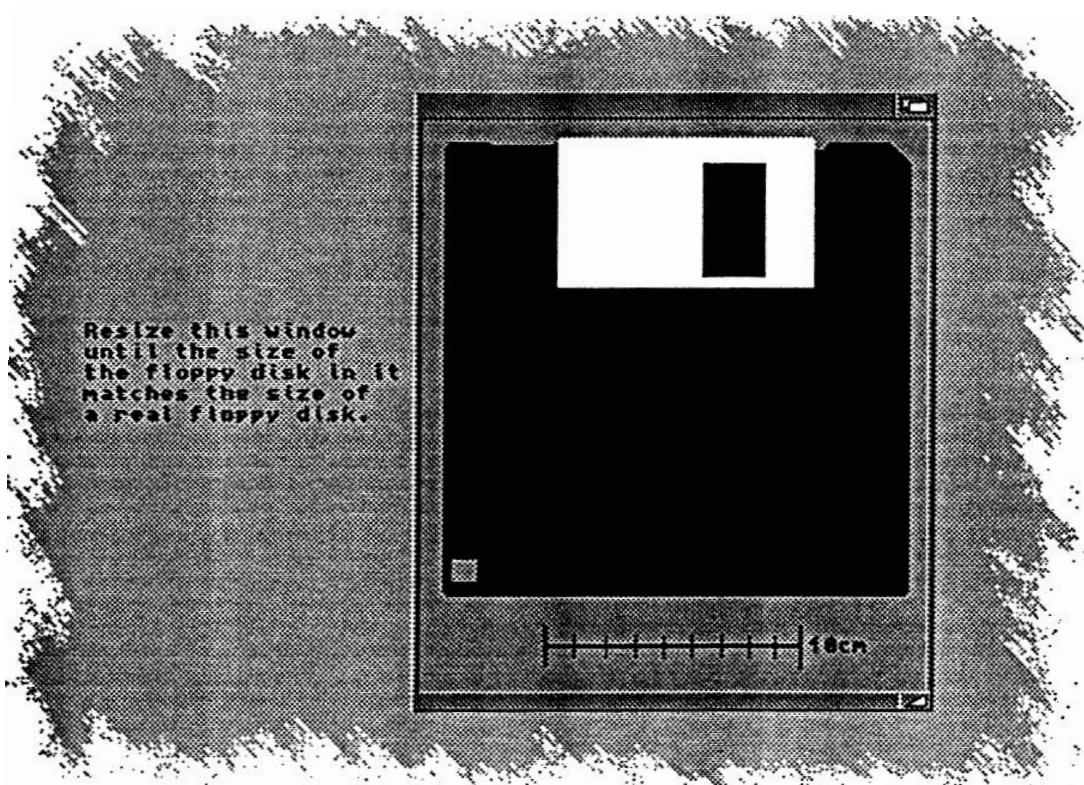
Page 18: menu header stacks and Amiga menu



Page 20: keyboard prefs



Page 20: mouse prefs



Page 22: monitor prefs - edit aspect ratio

(

(

(



Retargetable Graphics Specification

Copyright ©1992,1993 Commodore Electronics Ltd.

Proposed RTG system requirements:

- ECS or greater.
- Separate ROMS for ECS/AA and AAA via conditional compilation/assembly.
- '020 or greater required, at least for initial release.

A Graphics Driver

A graphics device driver is better viewed as a set of graphics routines being used by the graphics library, rather than as a library in its own right. Core differences are:

- ❑ The vector table size is determined at install time by `graphics.library`. This is done in order to allow the future extension by inclusion of new primitives in terms of older primitives, and to allow graphics to install its preferred default entry points.
- ❑ The device driver pointer is passed to the entry point in A5. GfxBase is kept in A6 for the execution of the function. Register conventions follow the normal standard (a0/a1 d0/d1 are scratch, return values in d0) unless otherwise specified.

Driver Functions

This section defines the functions needed to initialize a driver.

```
APTR CreateDriverA(struct TagList *tags)
```

creates a driver for a display device, and opens it. It returns either a pointer to the device base, or NULL if an error occurred.

TAGS:

DRV_InitVectorTable Points to a list specifying which default vectors should be replaced with driver-specific ones. (fmt= UWORD nentries, UWORD firstentry, ULONG &code)

DRV_CloneVectorTable Points to another graphics driver. Initial contents of the vector table are taken from this driver, instead of defaults.

The Graphics Database

This section describes the graphics database, and the role it plays in the Retargetable Graphics model.

Outline

The Amiga graphics system is capable of displaying many different types of display modes. As all these display modes have varying characteristics, such as colors, dimensions, etc, a database exists describing all the characteristics of all the modes; this removes the need for application writers to make assumptions about the display modes, especially as the characteristics can vary depending on machine configuration, chip type etc.

Each mode has a unique identifier ModeID, and is used as a key to the database functions.

The MonitorSpec

Although the MonitorSpec structure is publicly defined, it is highly unlikely that many developers will be looking at it. The only elements of the MonitorSpec that might be of use are the total_rows and total_colorclocks, which are used to calculate the monitor's refresh rates. The remainder of the structure is very specific to the Amiga Chip Set (in fact, so is the total_colorclocks, but we may have to live with that). However, there are still some elements that all the MonitorSpecs should have as a minimum.

Because of the fact that the MonitorSpec is mainly of importance to only the graphics library, it seems reasonable to redefine the structure as follows. I have tried to keep as many offsets in the same place as before as possible.

```
struct MonitorSpec
{
    struct ExtendedNode ms_Node;
    UWORD ms_Flags; /* driver specific flags */
    LONG MonitorID;
    APTR DriverBase; /* points to the driver base */
    UWORD total_rows; /* # of lines per field - same offset */
    UWORD total_colorclocks; /* line time in 280ns units */
    ULONG DisplayCompatible; /* how this mode coexists */
    UWORD min_row; /* at the same offset as before */
    APTR GfxPrivate; /* ours! (It's at the same place as a
    /* struct SpecialMonitor * is now).
    */
    UWORD OpenCount; /* At the same offset as before */
    struct List ModeIDs; /* Database entries hang off here */
    UWORD MoreFlags;
    struct Rectangle ms_LegalView; /* at the same offset as before */
    struct MonitorInfo *minfo; /* Pointer to a MonitorInfo structure
    /* that is inherited by all the ModeIDs
    /* of this monitor. */
    UBYTE DriverData[n]; /* driver specific data */
}
```

The MonitorSpec is now expandable, as 'n' is undefined. In the case of the native Amiga drivers, the DriverData contains much the same data as the V37 MonitorSpec.

The MonitorID is used to identify the monitor spec. For example, graphics would need to know if this is a MonitorSpec for the Amiga driver. In that case, it would know that it could look at the ms_Flags field.

New DataBase Functions

The bit layout of a monitor ID is as follows:

```
1nnn nnnn nnss ssuu
```

Bit 15 identifies a third party monitor type.

"nnnnnnnn" is the device code, which will be registered with us. "ssss" is the scan-rate subcode, which selects (for instance) different scan rates for the device. "uu" is the unit number of the device. The unit number is automatically incremented by CreateMonitorA (see below) when multiple units are added.

```
struct MonitorSpec *CreateMonitorA(struct NewMonitor *nm, struct TagList *tags)
```

creates a new MonitorSpec, and adds it to the list of monitors. It will return a pointer to a MonitorSpec, or NULL if error. This function will create a MonitorSpec large enough for the defined part plus the number of bytes in the driver specific area, and copy the driver specific data into the MonitorSpec.

```
struct NewMonitor
{
    ULONG    nm_MonitorID; /* Pointer to requested Monitor number. On
                          * successful return, this field will be
                          * set to the actual monitor number given.
                          */
    STRPTR   nm_MonitorName; /* Pointer to a string of the base monitor
                          * name. The string will be copied to the
                          * MonitorSpec.
                          */
    APTR     nm_DriverBase; /* Pointer to the DriverBase for this monitor */
    ULONG    nm_DeviceNumber; /* A number describing what physical device
                          * views with this monitor will appear on. 0
                          * Means the normal Amiga video output.
                          * This should be gotten with GetUniqueID().
                          */
    UWORD    nm_PreferredModeID; /* For preferences. The lower word of the whole
                          * ID that Overscan.prefs displays the editor
                          * screen in.
                          */
    UWORD    nm_TotalRows; /* Total Rows */
    UWORD    nm_TotalCClocks; /* Total ColorClocks per line
                          * (1 ColorClock = 280ns)
                          */
    UWORD    nm_Pad;
    struct Rectangle nm_LegalView; /* The LegalView rectangle */
    struct Point nm_ViewResolution; /* Monitor ticks-per-pixel */
};
```

TAGS:

AM_TextRange - pointer to a struct Rect32, for the TextRange overscan setting. Defaults to the NominalRange in the DimensionInfo of the PreferredModeID.

AM_GfxRange - Pointer to a struct Rect32 for the StdRange overscan setting. Defaults to TxtOScan.

AM_MS_Flags - Driver Specific flags. Defaults to 0

AM_MinRow - Defaults to 0.

AM_DataSize - The number of bytes in the DriverData

AM_DriverData - Points to driver specific data (requires the AM_DataSize tag).

AM_Compat - Compatibility - default is MCOMPAT_SELF

AM_DefaultViewPos - (for MonitorInfo) - defaults to (0, minrow)

AM_ViewPosition - (for MonitorInfo) - defaults to DefaultViewPos

Each call to CreateMonitorA() will cause the driver to be opened. That way it is possible to attach more than one monitor to a driver. For example, the AA driver will have all the Amiga monitors attached to it.

[NB - currently, the MonitorInfo has a pad[36] where the RawMonitorInfo has a Rect32 TxtOScan and a Rect32 StdOScan. We need to define these in displayinfo.h so the driver can add these too.]

If the monitor number is not the first monitor with that number (more than one identical board in the system), CreateMonitorA() will add a '1', '2' etc to the name. So, if this is the second "NTSC" monitor in the system (being on a plug-in AA card, with the native ECS chips being the first", CreateMonitorA() will make the name "NTSC1" for this monitor.

```
DisplayInfoHandle AddModeIDA(ULONG ModeID, struct TagList *tags)
```

This will create a new ModeID entry if this doesn't already exist, or add the specified data to an existing entry. If the specified data already exists for the ModeID, AddModeIDA() will return INVALID_ID as the handle. It is up to the driver to initialize the various entries properly. This is done either with subsequent calls to AddModeID() to initialize the entries for the first time, or SetDisplayInfoData() to modify entries.

TAGS:

AMID_Dims - Data points to an initialized struct DimensionInfo

AMID_Dispatch - Data points to an initialized struct DisplayInfo

AMID_Name - Data points to a string to name the ModeID

AMID_Driver - Data points to the DriverBase

AMID_DriverData - Data points to mode/driver specific data.

AMID_Dims, AMID_Driver and AMID_Dispatch must be specified for each mode. It is recommended that the entries for AMID_Dims, AMID_Dispatch, and optionally AMID_DriverData be specified when the Mode is first added.

DataBase Entries

For the most part, the DisplayInfo, DimensionInfo and NameInfo structures are unchanged. The MonitorInfo is slightly modified to reflect the graphics private RawMonitorInfo structure:

```
struct MonitorInfo
{
    struct QueryHeader Header;
    struct MonitorSpec *Mspc; /* pointer to monitor specification */
    Point ViewPosition; /* editable via preferences */
    Point ViewResolution; /* standard monitor ticks-per-pixel */
    struct Rectangle ViewPositionRange; /* fixed, hardware dependent */
    UWORD TotalRows; /* display height in scanlines */
    UWORD TotalColorClocks; /* scanline width in 280 ns units */
    UWORD MinRow; /* absolute minimum active scanline */
    WORD Compatibility; /* how this coexists with others */
    struct Rect32 TxtRange; /* \
    struct Rect32 GfxRange; /* | Replaces the UBYTE pad[36] */
    ULONG pad; /* /
    Point DefaultViewPosition; /* original, never changes */
    ULONG PreferredModeID; /* for Preferences */
    ULONG reserved[2]; /* terminator */
};
```

[All the current structures are terminated with a ULONG reserved[2], which was needed for the V37-V39 implementation of the database as a terminator. We may be able to dispose of these and save 8 bytes per structure per ModeID. Under V39, with all the monitors added, there are 264 ModeIDs. Not counting the NameInfos, that's 4 structures per entry, or 8448 bytes wasted as terminators! We will need to investigate the compatibility hit of s/w expecting so many bytes of data from a GetDisplayInfoData() call, but I don't see a real problem.]

There is, however, a small problem with the Txt/StdOScan values. These values are global to the whole Monitor, not just a specific ID. That means that the OScan values will need to be converted to values relevant for a specific ID. For example, on the standard PAL system, StdOScan may be 640x256 for hires, but 320x256 for Lores, and 1280x512 for SuperHires Laced. Therefore, the OScan values have to be stored in the greatest resolution possible, and scaled down.

The scaling factor comes from the DisplayInfo->Resolution value. GetDisplayInfoData() fills the Txt/StdOScan values of a DimensionInfo from the scaled Txt/StdOScan values in the MonitorInfo as:

```
dims->TxtOScan.MinX = (mntr->TxtOScan.MinX / disp->Resolution.x);
```

similarly for the other entries.

The struct VecInfo, which was introduced in V39 for graphics' private use should be replaced with a struct DriverInfo.

```
struct DriverInfo
{
    struct QueryHeader Header;
    APTR DriverBase; /* Points to the driver base for this
                      * ModeID.
                      */
    APTR Data; /* Points to driver specific info */
    UWORD pad[3]; /* alignment - could be removed */
    ULONG reserved[2];
};
```

We may also want to add another type, a struct `SpriteInfo`. The current information regarding sprites in the database is not too informative.

New display database information available:

Pixeltype ID - 4 character identifier for pixel-type. Identifies "weird" modes:

- 'HAM' upper two bits of pixel index determine operation. Lower n-2 bits are RGB or index data.
- 'EHB' High bit selects either `rgb[index]` or `rgb[index]/2`.
- 'TRUE' true-color. No palette.
- 'PLUT' pack lut mode.
- 'PRGB' packed rgb mode.

Whether or not a particular mode is "Displayable" at all. `DIPF_` flag.

A "skeleton" bitmap pointer is also available from the database for each mode. This gives you something to pass as the friend to `allocbitmap`.

The `DisplayInfo` structure will be updated accordingly.

```
struct DisplayInfo
{
    struct QueryHeader Header;
    UWORD NotAvailable; /* if NULL available, else see defines */
    ULONG PropertyFlags; /* Properties of this mode see defines */
    Point Resolution; /* ticks-per-pixel X/Y */
    UWORD PixelSpeed; /* approximation in nanoseconds */
    UWORD NumStdSprites; /* number of standard amiga sprites */
    UWORD PaletteRange; /* OBSOLETE - use Red/Green/Blue bits instead */
    Point SpriteResolution; /* std sprite ticks-per-pixel X/Y */
    UBYTE pad[4]; /* used internally */
    UBYTE RedBits; /* number of Red bits this display supports (V39) */
    UBYTE GreenBits; /* number of Green bits this display supports (V39) */
    UBYTE BlueBits; /* number of Blue bits this display supports (V39) */
    TEXT PixelType[4]; /* 4-character identifier for pixel type. */
    UBYTE pad2; /* find some use for this. */
    struct NaturalBitMap *FriendBitMap;
    /* Use as a Friend BitMap to pass to AllocBitMap()
     * when opening the first ViewPort of this ModeID.
     */
};

#define DIPF_NOT_DISPLAYABLE 0x00400000
```

Driver entry points:

The following functions are all related to Views and ViewPorts, and need to be vectored by the driver. Most of these take the same parameters as their LVO counterparts. Differences are described.

`/drv_ObtainDBufInfo(vp,dbi)` - attach and initialize any driver specific information for double buffering.

`/drv_ReleaseDbufInfo(db)` - do any double buffering cleanup

`/drv_ChangeVPBitMap()`

`/drv_FreeVPortCache()` - this frees up any information cached by the driver at `MakeVPort()` and `MrgCop()` time. RTG equivalent of `_LVOFreeVPortCopLists()`.

/drv_RefreshColors() - Updates all the colors for the passed ViewPort from the ColorMap.
Should query colors using GetRGB32.

/drv_LoadView() - Load the view. The transition should be as clean as possible.
drv_LoadView(NULL) should blank the screen.

/drv_UnloadView() - tells the driver that the it's View is no longer active, and can thus free no longer needed resources.

/drv_MakeVPort() - Builds up all the intermediate data it needs to display a single ViewPort.

/drv_MakeView() - Takes all the intermediate data of all the ViewPorts of the View, to build the final data it needs to show the whole View. RTG equivalent of **_LVOMrgCop()**.

CopLists and CprLists allocated by MakeVPort and MrgCop will be tagged with respect to device, so that FreeCopList and FreeCprList can work.

/drv_FreeCopList(cop) - free cache information from MakeVPort.

/drv_FreeCprList(cpr) - free view info from MrgCop.

/drv_MoveSprite()

drv_ChangeSprite()

/drv_UpdateVPort() - should take account of changes to BitMap pointer, rxyoffset. RTG equivalent to **_LVOScrollVPort()**.

/drv_WaitBeam() - used for WaitBeam and WaitBOVP.

/drv_AsyncWaitBeam() - async wait beam. Both WaitBeams return -1 if syncing was impossible.

/drv_CalcIVG()

/drv_WaitTOF() - new entry will need to take a View

?/drv_VBeamPos() - new entry will need to take a View?

/drv_GetSpriteInfoA() - returns the number of sprites that can be displayed on the passed ViewPort.

/drv_ValidVTAGSupported - checks whether a particular VideoControl tag is supported by the device.

/drv_VideoControl() - graphics first handles system videocontrol() functions (like **ATTACH_CM_SET**), and then passes the tag list onto **drv_VideoControl**. **drv_VideoControl** should invalidate the **VTAG_IMMEDIATE** variable.

Definitions of New Driver Functions

Driver entry points:

(/ indicates mandatory entries) all functions are passed `a5=Struct GfxDeviceDriver *` and `a6=struct GfxBase *`, unless otherwise noted. Functions must follow standard library function calling conventions unless otherwise noted.

```
void drv_RefreshColors(struct ViewPort *vp);
```

On the Amiga chips, this updates the color instructions in the intermediate and system copperlists with the contents of the ViewPort's ColorMap. On a 3rd party device, this will update the intermediate data and the hardware registers responsible for colors with the contents of the ViewPort's ColorMap.

```
void drv_KillView(struct MonitorSpec *mspc);
```

This tells the MonitorSpec that was associated with a View that it is no longer being displayed. This allows the driver to free any resources it may need only while the View is active. For example, the A2024 driver removes the vertical blank interrupt handler when it is no longer active with it's KillView() entry. This function should only be called by LoadView().

New Layers LVO

```
inout=FindLayerCR(struct RastPort *rp, LONG x, LONG y,  
    struct Rect32 *cliprect, struct Point32 *offset,  
    struct BitMap **bm)
```

Layer must be locked. Find a point in the layer. Set *cliprect to the bounds of the cliprect. Set *bm to point to the correct bitmap. If the point is outside of the cliprect list, set *cliprect to an exclusion rectangle if the pointer is non-null.

Layers will also have to grow support for the new rastport region stuff, both in FindLayerCR and DoHookClipRects.

Definitions of new LVOs

These new LVOs also have corresponding drv_ functions.

```
void NewWaitTOF(struct View *view, [struct Message *msg]);  
void drv_WaitTOF(struct View *view, [struct Message *msg]);
```

This will wait until the beam has reached the top of the display. ?how do you find the view pointer for your intuition screen?

```
LONG NewVBeamPos(struct View *view);
LONG drv_VBeamPos(struct View *view);
```

This one is still only of use when the caller knows it won't be preempted. Also, the current `_LVOVBeamPos()` in `graphics.library` doesn't work properly anyway. This new one will! Returns -1 if board is incapable of doing it.

```
BOOL /drv_WaitBeam(struct View *v, struct ViewPort *vp, ULONG beamposy);
```

Return when the beam is at the given position (approximately). May break a forbid. May use polling. If the device can't beam synchronize, return false. Can automatically adapt to CPU speed?

```
BOOL /drv_ASyncWaitBeam(struct View *v, struct ViewPort *vp, ULONG beamposy, struct
Message *msg);
```

Reply the message when the beam reaches a certain position. May fail.

```
void GetSpriteInfoA(struct ViewPort *vp, struct TagList *tags);
void drv_GetSpriteInfoA(struct ViewPort *vp, struct TagList *tags);
```

Many applications need to know how many sprites they can use in a given ViewPort, but the database only tells them how many sprites the hardware can support at most. This new call should return more useful information.

TAGS:

MS_Origin - How many sprites can be used when this ViewPort is at it's origin position. `ti_Data` points to a ULONG to store the result (yes, **real** tags in graphics).

MS_WorstCase - The worstcase number of sprites that would be available (eg on the Amiga, worst case would be while scrolling the ViewPort). `ti_Data` points to a ULONG to store the result

MS_WithTricks - The worstcase number of sprites that would be available if tricks were pulled. eg, on the Amiga, we could drop the bandwidth to increase the number of sprites. `ti_Data` points to a ULONG to store the result

MS_ReUse - If the device supports reusing of sprites, then the result is the number of lines needed between each sprite reuse. If sprites cannot be reused, the result is -1. eg, the Amiga would return 1. `ti_Data` points to a ULONG to store the result

Getting Started

Call `GetMonitorNumber()` with the Monitor number allocated by C= for your card. Use the resultant monitor number in all the `ModeIDs` for the monitor.

Create the Driver.

Initialize the `MonitorSpec`.

Call `CreateMonitor()`.

Initialize and call `AddModeID()` for each mode in the monitor.

Easy!

Rough Example

```
#define MONITOR_NUM 0x40010000
#define MONITOR_NAME "MyMonitor"
#define MONITOR_PREFERREDID ???
#define MONITOR_FLAGS 0
#define MONITOR_TOTROWS ???
#define MONITOR_TOTCC ???
#define MONITOR_COMPAT MCOMPAT_NOBODY
#define MONITOR_MINROW ???
#define MONITOR_DATASIZE ???

struct MonitorSpec *mspc;
APTR DriverBase;
ULONG MonitorNum = MONITOR_NUM;
UBYTE MonitorData[MONITOR_DATASIZE];

struct TagItem DriverTags[] =
{
    (DRV_InitVectorTable, VectorTable),
    (TAG_END),
};

struct TagItem MonitorTags[] =
{
    (AM_MS_Flags, MONITOR_FLAGS),
    (AM_Compat, MONITOR_COMPAT),
    (AM_MinRow, MONITOR_MINROW),
    (AM_DataSize, MONITOR_DATASIZE),
    (AM_DriverData, MonitorData),
    (TAG_END),
};

struct NewMonitor nm =
{
    &MonitorNum,
    MONITOR_NAME,
    NULL,
    NULL,
    MONITOR_PREFERREDID,
    MONITOR_TOTROWS,
    MONITOR_TOTCC,
    NULL,
    /* LegalView Here. */,
    /* View Resolution here */,
};

/* Create the Driver */
if (DriverBase = CreateDriverA(DriverTags))
{
    /* initialize the NewMonitor */
    nm.Driverbase = DriverBase;
    nm.DriverNumber = GetUniqueID();

    /* Add the Monitor */
    if (mspc = CreateMonitorA(&nm, MonitorTags))
    {
        /* CreateMonitorA() may have assigned a different monitor number
```

```

        * and name.
        */
        printf("Monitor Number assigned is 0x%lx\n", MonitorNum);
        printf("Monitor name is %s\n", mspc->ms_Node.xln_Name);

        /* for each ModeID in the monitor,
        * create the data, and call AddModeID()
        */
    }
    else
    {
        /* Whoops. Clean up time. */
        CloseDriver(DriverBase);
    }
}

```

DriverBase

The DriverBase works in much the same way as a LibraryBase. The positive offsets from the base are for driver data, and the negative offsets are function pointers.

ViewPortExtra

The ViewPortExtra structure was extended for V39.

```

struct ViewPortExtra
{
    struct ExtendedNode n;
    struct ViewPort *ViewPort;          /* backwards link */
    struct Rectangle DisplayClip;        /* makevp display clipping information
    /* These are added for V39 */
    APTR VecTable;                      /* Private */
    APTR DriverData[2];
    UWORD Flags;
    Point Origin[2];                   /* First visible point relative to the
                                        DClip.
                                        One for each possible playfield.
                                        */
};

```

VecTable points to the same VecTable that the DriverInfo for the ViewPort's mode points to. DriverData[] are pointers to driver specific data, and are free for the driver to use. Flags is for the system's.

We no longer need the Origin[]. We could remove these, and replace them with

```

APTR DriverBase;
ULONG DriverFlags;

```

How to get at the driver vector table from a given structure

RastPort	through the BitMap
BitMap	if bm_BytesPerRow odd, than has DriverPtr, else Default.
View	Either the topmost viewport, or the ViewExtra
ViewPort	through the ColorMap or RasInfo->BitMap
ColorMap	through the ViewPortExtra
ViewPortExtra	contains a pointer to it.

RTG Rendering

```
struct StandardBitMap
{
    UWORD   BytesPerRow;    /* odd means a NaturalBitMap */
    UWORD   Rows;          /* height */
    UBYTE   Flags;          /* reserved for graphics */
    UBYTE   Depth;         /* bits per pixel up to 255 */
    UWORD   pad;           /* reserved for graphics */
    PLANEPTR Planes[];      /* plane pointers */
};
```

If the `bm_pad` is equal to a magic value, then the flags field is valid. Flags include interleaved, true-color, fastmem, etc. A colormap can be associated with a `StandardBitMap` (stored in `Planes[Depth]`) via `SetBitMapAttrs`. This is so that standard indexed bitmaps can be blitted to true-color screens. When a bitmap is passed as a friend to `AllocBitMap`, its `ColorMap` is installed into the new bitmap.

An embedded `StandardBitMap` is created by calling `AllocBitMap` with the appropriate flags and then calling `SetBitMapAttrs` to attach the planes.

A `StandardBitMap` is distinguished from a `NaturalBitMap` by bit 0 of `BytesPerRow`. Bit 0 is set for a `NaturalBitMap`.

A true-color bitmap has depth/3 planes for each gun. (or maybe depth?)

```
struct NaturalBitMap    // a bitmap for a foreign device driver
{
    UWORD nb_GfxFlags;    /* graphics flag. Bit 1 always set */
    UWORD nb_Rows;        /* height */
    UBYTE nb_Flags;       /* true-color, etc. */
    UBYTE nb_Depth;       /* bits per pixel, up to 255 */
    UWORD nb_pad;         /* set to magic number */
    /* fields after this do not match a normal bitmap structure */
    ULONG nb_NewFlags;    /* flags for gfx use. */
    struct GfxDeviceDriver *nb_Driver; /* pointer to device driver */
    struct ColorMap *nb_AssociatedColorMap; /* colormap associated with this display bitmap */
    ULONG nb_FutureExpansion[2]; /* future for graphics */
    driverdata[];         /* variable length device information */
};

struct RastPort
{
    struct Layer *Layer;
    struct BitMap *BitMap; /* could now be a NaturalBitMap */
    UWORD *AreaPtrn;       /* ptr to areafill pattern */
    struct TmpRas *TmpRas; /* obsolete for non-amiga drivers, but should definitely not be re-used */
    struct AreaInfo *AreaInfo; /* AreaInfo ptr for all devices */
    struct GelsInfo *GelsInfo; /* yuck, gross. why? */
    UBYTE Mask;            /* driver private for >8 bit drivers */
    BYTE FgPen;            /* driver private for >8 bit drivers */
    BYTE BgPen;            /* driver private for >8 bit drivers */
    BYTE AOLPen;           /* driver private for >8 bit drivers */
    BYTE DrawMode;         /* standard drawmode */
    BYTE AreaPtSz;         /* 2^n words for areafill pattern. */
    BYTE linpatcnt;        /* current line drawing pattern preshift */
    BYTE dummy;            /* graphics private. use to store last charfor kanji */
    UWORD Flags;           /* miscellaneous control bits. Graphics private */
    UWORD LinePtrn;        /* 16 bits for textured lines */
    WORD cp_X, cp_Y;       /* driver private. Drivers supporting 32 bit coordinates will want to store these elsewhere */
    UBYTE minterms[8];     /* driver private!! */
    WORD PenWidth;         /* reserved for graphics.library */
    WORD PenHeight;        /* reserved for graphics.library */
};
```

```

struct TextFont *Font;      /* current font address */
UBYTE AlgoStyle;           /* the algorithmically generated style */
UBYTE TxFlags;             /* text specific flags */
UWORD TxHeight;           /* text height */
UWORD TxWidth;            /* text nominal width */
UWORD TxBaseline;         /* text baseline */
WORD TxSpacing;           /* text spacing (per character) */
APTR *RP_User;            /* user data */
ULONG longreserved[2];     /* reserved for graphics */
UWORD wordreserved[7];     /* reserved for driver */
UBYTE reserved[8];        /* reserved for graphics */
/*
    gfx expansion bytes: 24
    extra bytes available to 8 bit driver: 23
    extra bytes for 24 bit driver: 17
    extra bytes for 24 bit driver with 32 bit coordinates: 13
*/

```

Low-Level Rendering Operations

```
BOOL /drv_GetBitMapAttr(bm,attr,&data)
```

return the given attribute for a bitmap. returns whether or not the driver knows the given attribute.

```
BOOL /drv_SetBitMapAttr(bm,attr,data)
```

set the given attribute for a bitmap. returns whether or not the driver knows the given attribute.

```
struct BitMap * /drv_AllocBitMap(size, sizey, depth, flags, friend)
```

Allocates a bitmap.

```
/drv_ReadPixData(bm,x,y,w,h,&buffer)
```

Read pixels from a rectangle in a bitmap. The storage size will be (bm->depth+7)/8 bytes per pixel. The driver may store the data in whatever manner it chooses, as graphics.library will always treat it as whole pixels.

```
/drv_WritePixData(bm,x,y,w,h,,&buffer)
```

Write pixels to a rectangle in a bitmap. The storage size will be the same as ReadPixData. ReadPixData and WritePixData can be used to implement all blits and scaling blits.

```
/void drv_WaitBlit(void)
```

Wait for any previously queued graphics operations to complete. May NOT break a FORBID!!. WaitBlit on the standard HW.

```
/void drv_Flush(void)
```

Wait for any previously queued graphics operations with this driver to complete. Must be called before FreeBitMap. MAY break a FORBID!!. WaitBlit on the standard HW.

```
/drv_SetRastBM(struct NaturalBitMap *bm, struct RastPort *rp, minx, maxx, miny, maxy, color)
```

set the given rectangle in the bitmap to the given indexed color. When blitting to truecolor, will do a color lookup.

```
/drv_BltTemplateBM(WORD *src, srcx, srcmod, DestRPort, destx, dsty, sizex, sizey, destbm)
```

Blit template to a bitmap. Should obey RastPort parameters.

```
/drv_MatchTemplateBM(rp, struct BitMap *bm, srcx, srcy, dstx, dsty, width, height)
```

Fill plane 0 of the specified rectangle of bm with 1's wherever the pixels match the APen setting of the rastport. Used by Flood and highlight mode.

```
struct DDA {
    FUNCPTR AdvanceDir[8] ; ptrs to functions for advancing
                        ; U UR R DR D DL L UL
    FUNCPTR DrawAdvanceDir[8]
                        ; ptrs to functions for drawing and advancing
                        ; U UR R DR D DL L UL
    FUNCPTR Move         ; move with no draw. Resets based on BitMap (a1).
    ULONG driverdata[?]; ; DDA context. The DDA context also consists of
                        ; a0 and d0.
}

/err=drv_InitDDA(struct DDA *dda, struct RastPort *rp, struct BitMap *bm, x, y)
```

Initialize for pixel drawing, based upon rastport settings. Must fill in vector pointers. Can use An and Dn for temps across the lifetime of a DDA. Multiple DDA's must not interfere with each other (meaning that typically they will have locking equivalent to OwnBlitter)

```
/drv_FinishDDA(struct DDA *)
```

free up any resources associated with a DDA.

```
/drv_WritePixelBM(struct BitMap *bm, struct RastPort *rp, x, y)
```

Write a pixel into a bitmap, using current context settings in rastport.

```
ULONG /drv_ReadPixelBM(struct BitMap *bm, struct RastPort *rp, x, y)
```

Read a pixel from a bitmap. Straight ReadPixel is not too useful on a true-color device. Most Set/Get rastport calls map to direct drv_xxx calls, and are mandatory.

```
/drv_ReadPixelRGB8ArrayBM(bm, xl, yl, w, h, &rgb32)
```

Get 8-bit rgb values for a pixel array. Even works in HAM mode.

```
/drv_WritePixelRGB8ArrayBM(bm, xl, yl, w, h, &rgb32)
```

Write 8-bit rgb values for a pixel array. Only works for true-color modes.

```
/drv_WritePixelArrayBM(bm, xl, yl, w, h, &rgb32)
```

Write (depth+7)/8-bit index values from a pixel array.

```
/drv_ReadPixelArrayBM(bm, xl, yl, w, h, &rgb32)
```

Read (depth+7)/8-bit index values from a pixel array.

```
/drv_Move(rp, x, y)
```

Move the cursor position.

Medium Level Primitives

```
drv_DrawLine(rp, bm, x1, y1, x2, y2)
```

Draw a line, unclipped

```
drv_DrawClippedLine(rp, bm, x1, y1, x2, y2, &rect32)
```

Draw a line, clipped to the given rectangle. Must use perfect raster clipping, and handle the pattern shift properly.

```
drv_Bltxxx(srcbm, srcx, srcy, destx, desty, width, height, msk, op)
```

BitBltMap family of functions. The default BitBltMap demultiplexes the various combinations and calls one of the following:

```
NaturalToNatural  
StandardIndexedtoNatural  
StandardTrueColorToNatural  
NaturalToStandardIndexed  
NaturalToStandardTrueColor
```

```
drv_BltMaskBitMapBM(srcbm, srcx, srcy, destbm, destX, destY, sizeX, sizeY, op, bltmask)
```

Perform masked blit. Default entry will do multiple BitBltMaps.

```
drv_FilledEllipse
```

draw a filled ellipse, using areafill rules. This will be called by AreaEnd when there is only one ellipse in the area buffer

```
drv_FilledPoly
```

draw a filled (potentially concave) polygon.

```
/drv_BltPatternBM(rp, mask, xmin, ymin, xmax, ymax, maskbpr, &bm)
```

Do a pattern blit, using rastport settings.

```
/drv_FillYLRBM(rp, xoffset, yoffset, &ylrarray, &bm, &cliprect, patxoffset, patyoffset)
```

Fill a series of horizontal lines, using pattern fill rules. &cliprect may be null (meaning no clipping). the ylrarray is an array of ULONG triplates, terminated by -1. Many of the higher level calls which call this will generate the array in top to bottom order, so it is worth optimizing for this case (though you can't depend on it).

High Level Primitives

```
drv_Flood(rp,mode,x,y)
drv_BltBitMap()
drv_DrawEllipse(rp,xcenter,ycenter,a,b)
drv_ReadPixelRGB32(rp,x,y,&rgb32)
```

Read pixel data out. handles indexed color or true-color. handles HAM! Rp Entry is needed for HAM support through layers.

WritePixel example:

```
#define BMTODRIVER(bm) \
    (bm->BytesPerRow & 1)? \
    ((struct NaturalBitMap *)bm)->bm_GfxDriver \
    :GBASE->DefaultDriver

CALLOFFBITMAP is impossible to #define in C, but it doesn't matter for this example.

WritePixel(rp,x,y)          // _LVOWritePixel points here.
{
    if (rp->Layer)
    {
        struct BitMap *found_bitmap;
        COORD32 xyoffsets;
        LOCKLAYER(rp->Layer)
        if (FindLayerCR(rp,x,y,NULL,&xyoffsets,&found_bitmap))
        {
            CALLOFFBITMAP(found_bitmap,OFS_WRITEPIXELBM)(rp->BitMap,rp,
                x+xyoffsets.x,y+xyoffsets.y,BMTODRIVER(found_bitmap));
        }
        UNLOCKLAYER(rp->Layer);
    }
    else
    {
        CALLOFFBITMAP(found_bitmap,OFS_WRITEPIXELBM)(rp->BitMap,rp,x,y,
            BMTODRIVER(found_bitmap));
    }
}

void drv_WritePixelBM(register __a0 struct myBitMap *mybm,          // NaturalBitMap
ptr
                      register __a1 struct RastPort *myrp,
                      register __d0 LONG x, register __d1 LONG y,
                      register __a5 struct MyLibBase *MyLibBase)
{
    ObtainSemaphore(MyLibBase->BlitSemaphore);
    /* not really needed for atomic operation */
    UBYTE *pixeladr;
    UBYTE pixdata;
    UBYTE msk=rp->Mask;

    pixdata=(rp->DrawMode & INVERSEVID)?rp->FgPen:rp->BgPen;
    pixdata &= msk;
    pixeladr=mybm->BytesPerRow*y+x;
    switch((rp->DrawMode) & ~INVERSEVID) /* simple chunky device.
                                         Uses normal DrawMode
                                         in rastport for SetDrMd
                                         storage */
    {
        case JAM1:
        case JAM2:
            *pixeladr= *(pixeladr & ~msk) | pixdata;
            break;
        case COMPLEMENT:
            *pixeladr ^= msk;
    }
}
```

True color blit rules:

Bitmaps allocated as friends of other bitmaps inherit their colortables.

indexed->indexed: no translation

indexed -> true color

Src bitmap	dest bitmap	
=====	=====	
no table	yes table	uses dest table
yes table	yes table	assumes src table
yes table	no table	uses src table
no table	no table	uses default table for source

true-color-> true color

no colortable needed

General Graphics Enhancements & Misc RTG Issues

New graphics videocontrol tag: VTAG_INTERMED_UPDATE : When this is set for a viewport, graphics calls which change the copper lists need not update the information cached by MakeVPort (the intermediate copper lists). Only the information (hardware copper lists) generated by MrgCop need be updated. This applies to LoadRGB32, ScrollVPort, ChangeUCL, VideoControl with VTAG_IMMEDIATE, and ChangeVPBitMap. Basically you are telling graphics that you will not be MrgCop'ing again without remaking the viewport. Should help games/multimedia a lot.

Requires intuition support to protect screens from RethinkDisplay.

VideoControl will either stop barfing on bad tags, or will be replaced.

VideoControl will grow a new tag for turning "ScreenFlash" on and off. This will be used for ScreenBeep().

New alloc/free structures will be handled via GfxNewTags and GfxFree to stop LVO overpopulation.

```
GfxNewTagList (gfxnodetype, tags)
```

aliased to GfxNew.

Point transformations:

```
struct Transformation *GfxNewTags(GFXTYPE_TRANSFORM, tags)
```

allows transformation from real-world units to screen units, rotation, scaling, translation, and shearing. Transformations can be concatenated by passing an old transformation.

```
TransformPoints(struct Transformation *tf, &srcpts, &dstpts, n)
```

src and dest can be same.

```
InverseTransformPoints(struct Transformation *tf, &srcpts, &dstpts, n)
```

src and dest can be same.

graphics clipping will be made robust out to 16 bits (for the motherboard driver). The actual primitive implementation will be robust to 32 bits for other drivers.

?What to Do About HAM10?

GetRGB32 will return the system default colors if the passed ColorMap is NULL. This is for support for true-color bitmaps when there is no colormap attached.

PenWidth and PenHeight will be supported for Draw, DrawEllipse, and DrawBezier.

SetRPAtrrs can be used to set the true-color to render in on true-color devices.

```
FillYLR(rp, xoffset, yoffset, &ylrarray)
```

Fill a series of horizontal lines, using BltPattern rules. Calls drv_FillYLRBM(rp, xoffset, yoffset, &ylrarray, &bm, &cliprect).

```
SetBitMapAttrrsA(bm, attrrs)
```

Change bitmap attributes. Will be used to install a colormap into a bitmap.

```
DrawBezier(rp, &points)
```

Draw a bezier curve (or a series of them?)

```
ElipiticalArc(rp, points)
```

Draw an elliptical from 3 points.

```
?RemapBitMap(&srcbm, &destbm, x, y, w, h, &colorpens, ...)?  
RemapBitMapRastPort(&srcbm, &destrp, x, y, w, h, &colorpens)
```

Remap a bitmap. Can convert true-color/ham to indexed. Can convert indexed to true-color, etc. ColorPens is a handle to a penarray to free via GfxFree.

?AreaBezier?

ObtainBestpen will allow specification of a color to avoid. This is to avoid the catastrophe of black-on-black text. It will also gain the ability to give back "n" colors at once (each with an associated weight), which will improve remap quality.

Bitmaps will work in virtual memory / fast ram

SetRPAtrrs can be used to set a clipping region and (maybe) origin for a RastPort. This is similar to InstallClipRegion, but does not modify the layer's clip rectangles, and works even if they change out from under it. Installing the region will be inexpensive, but drawing through it will be slower than the InstallClipRegion method. The other advantage of the rastPort clipping region is that it works on non-layered rastports, and that there can be multiple independent ones per layer. I would like to re-

implement regions for this release, in order to:

- (a) Use exec memory pools
- (b) keep them y/x sorted
- (c) Add the (future or present) capability for bitmapped or other types of regions.
- (d) make it possible in the future to generate them by calling graphics primitives (like on the Mac).
- (e) allow sharing of spans and rectangles between multiple regions via reference counting.

SetRPAtrrs will also be able to set the pen width and height and other style tags.

Smart fonts:

A Smart font knows how to render itself. Hanging off of the font extension is a pointer to a vector table. The following vectors are present:

```
OpenFont()
CloseFont()
DrawText(textparams)
    inittext(params)
    RenderTemplate(temp, rp, string, bounds)
    donetext(params)

TextLength()
TextFit()
TextExtent()
```

and also vectors for diskfont.library:

```
MakeScaledFont(params)

SmoothPoly(rp, &points, &colors, npoints, [lookuptable])
```

Gouraud shade a convex polygon. lookuptable is for indexed-color systems, and is optional. If no lookuptable is specified on an indexed color system, actual color indices will be interpolated. A simple ordered dither is supported.

```
BOOL RectInRegion(struct Region *, struct Rectangle *)
```

Returns true if there is any possible intersection between the rectangle and region.

```
BltBitMapTags(srcbm, x, y, destbm, x, y, width, height, tags....)
BltBitMapRastPortTags(srcbm, x, y, destrp, x, y, width, height, tags....)
```

New style blit. Supports AAA and true-color arithmetic modes: defaults to copy.

BBMTAG_Mode:

BBM_ADD	constants can be added via subtraction.
BBM_ADDS	add with saturate
BBM_SUB(S)	
BBM_BLEND	blending %
BBM_OR	
BBM_AND	
BBM_COPY	
.etc.	

BBMTAG_MaskBitMap, &standardbitmap. Arbitrary depths are supported for blending.

A device driver can be written which would act to store displaylists, which would create the ability for simple printing, resolution independence, etc.

DeviceSpecificFunction(funcnum, tags)

funcnums 0-7ffffff reserved for standardization.

IsSupported(funcnum, bitmap)

ChangeUCL:

```
***** ChangeUCLA *****
*
*  NAME
*  ChangeUCLA -- Modify a viewport's user-copperlist
*  ChangeUCL  -- varargs stub fro ChangeUCLA
*
*  SYNOPSIS
*  status= ChangeUCLA(vp, regnum, regmask, tags)
*             a0      d0      d1      a1
*
*  LONG ChangeUCLA(struct ViewPort *vp, ULONG regnum,
*                  ULONG regmask, struct TagItem *);
*  LONG ChangeUCL(struct ViewPort *vp, ULONG regnum,
*                  ULONG regmask, tag1,...);
*
*  FUNCTION
*  To Modify (or optionally read) register values from a viewport's
*  copper list.
*
*  INPUTS
*  vp      - pointer to a viewport.
*  regnum  - address of custom chip register to modify MOVE for in
*            the copper list
*  regmask - mask of bits to be changed.
*  tags    - standard tag list. The following tags are currently supported:
*
*            UCL_ReadValue: Reads the current value from the copper list into
*                           the long variable pointed to by ti_Data.
*            UCL_WhichWait: Specifies how many CWAITS in your user copper list
*                           to skip over before looking for the register. Defaults
*                           to 0.
*            UCL_WhichReg: Specifies how occurrences of the given register
*                           to skip over (after UCL_WhichWait is satisfied). Defaults
*                           to 0. It will stop when it finds the corrent occurrence
*                           of the register, or at the next wait.
*            UCL_NWaits: specifies how many CWAIT/CMOVE groups to apply the
*                           given changes to.
*            UCL_NRegs: specifies how many occurrences of the appropriate register
*                           to change in each CWAIT/CMOVE group.
*            UCL_CachePtr : Allows The system to cache the locations of
*                           user-copper list instructions. You should pass a pointer to
*                           a void * pointer in the ti_Data. When your program is done, it
*                           should FreeVec the cache information pointed to by the
*                           pointer.
*
*  RESULT
*  0 = function not implemented (value returned by V39)
*  <0 = an error occurred
*  >0 = success.
*
*  NOTE:
*  This function is hardware dependent, and only works on standard Amiga-
*  compatible chips.
*****
```




Introduction to CAMD

by Dan Baker

CAMD, the Commodore Amiga Multimedia Driver, is a shared library that provides for real-time synchronization and data sharing between multimedia applications. Any application that requires the timed coordination of visual or audio events can use CAMD instead of creating a custom solution. This article presents an overview of CAMD and some examples of how to use it. In order to understand all the material presented here, you need to have some experience writing applications for the Amiga and also be familiar with the MIDI data standard.

The main components of CAMD are two shared, run-time libraries:

- ☐ `realtime.library` - provides the timing facilities, functions and structures for coordinating audio-visual information in real time.
- ☐ `camd.library` - provides data sharing and transmission functions using the MIDI standard.

The `realtime.library` handles timing while the `camd.library` handles the movement of data between applications and devices. Even though there are two separate libraries, in this article we refer to them together as CAMD for convenience.

Goals of CAMD

The goals of CAMD are simple:

- ☐ To encourage development of multimedia and music applications by providing system-level tools
- ☐ To enable many multimedia and MIDI applications to operate concurrently in an orderly fashion

Until CAMD, Commodore never provided system support for MIDI applications, so each application developer had to come up with their own custom solution. This has hindered the development of MIDI software on the Amiga. Similarly, because each application had to provide its own custom solution, MIDI applications typically did not work well together and could not take full advantage of the Amiga's multitasking capabilities.

CAMD is designed to solve these problems. Many of the design issues MIDI application developers have to deal with are the same as for multimedia applications, especially synchronization. So the facilities of CAMD have been made general enough to serve multimedia needs as well.

Timing and the *realtime.library*

One of the most difficult issues facing music and multimedia application developers is timing. On the Amiga, the two 8520 CIA chips provide the best source of timing information for applications, however, these can be hard to use because the interface provided by the system is arcane.

Realtime.library provides a convenient, higher-level interface to the 8520 CIA timers that is easy to use. Realtime.library also provides for the distribution of timing pulses to an unlimited number of client applications on a priority basis, thus supporting multitasking in the most robust manner possible.

Conductors and PlayerInfos

The realtime.library uses the Conductor structure to manage timing. There can be any number of Conductor structures, each of which represents a separate and independent timing context (i.e., a group of applications that wants to be synced together).

Each Conductor can have one or more client applications. A second structure called a PlayerInfo, is set up by each client application that wants to get timing information from the Conductor. There is typically one PlayerInfo for each task that wants to get timing information (a task could have more than one but this would be unusual).

Both the Conductor and PlayerInfo structures follow the conventions of Release 2. You never create these structures by allocating and initializing them yourself. The system provides the functions to do this for you. Also the structures are read-only. To change the fields within these structures, use the system-provided functions and tags.

An application will usually create a single PlayerInfo structure and then attach it to a Conductor. If the Conductor does not yet exist, it will be created by the system. To create a PlayerInfo structure you call CreatePlayer():

```
struct PlayerInfo *pi = CreatePlayer(Tag tag, ... );
```

This call takes a list of tag items that describe the attributes of the PlayerInfo structure you want to create. (A complete list of all the tags available can be found in the Autodoc for

SetPlayerAttrs(). It returns a pointer to the **PlayerInfo**. Here's a fragment showing how to set up a **PlayerInfo**:

```
struct Library      *RealTimeBase = NULL;
struct PlayerInfo   *pPlayerInfo  = NULL;

RealTimeBase=OpenLibrary( "realtime.library", 0L );
if(RealTimeBase)
{
    pPlayerInfo = CreatePlayer( PLAYER_Name,      "My_player",
                                PLAYER_Conductor, "My_conductor",
                                TAG_END );

    if(pPlayerInfo)
    {
        /* Your real-time application goes here... */

        DeletePlayer(pPlayerInfo);
    }
    CloseLibrary(RealTimeBase);
}
```

In the code above, a **PlayerInfo** will be created with the name of "My_player". It will be attached to the **Conductor** structure named "My_conductor". If a **Conductor** structure named "My_conductor" does not already exist, the system will create one. Other applications could also attach their **PlayerInfos** to "My_conductor".

When your application finishes, you should delete any **PlayerInfos** you created by calling **DeletePlayer()**. The **Conductor** will be automatically deleted by the system (however this won't happen until **all** the **PlayerInfos** attached to a **Conductor** are deleted).

Once you have a **PlayerInfo** and **Conductor** set up, you can obtain timing pulses for your application. Timing pulses come from the 8520 CIA chips (whichever one is available) and are passed to your application through the **Conductor**.

Getting Clock Ticks

The **realtime.library** uses a tick frequency of 600 Hz so clock pulses are delivered approximately every 1.66 ms. There are two ways to get this timing information:

- ☐ An alarm signal
- ☐ A clock tick callback hook

You can ask **realtime.library** to signal your task at some future time by using its alarm facility. This allows you to operate asynchronously. For instance, you could start a group of MIDI notes playing and set the realtime alarm to signal you when they should be stopped, then go on to some other job such as preparing the next group of notes before calling **Wait()** on the alarm signal.

The fragment below shows how to set up the `realtime.library`'s alarm clock to signal the calling task at time = 1000 ticks (the fragment assumes that the `realtime.library` is already open).

```

LONG          midiSignal,res;
BOOL          timerr;
struct PlayerInfo *pPlayerInfo=NULL;

/* This fragment assumes that realtime.library is already open. */
midiSignal = AllocSignal(-1L); /* Allocate a wake-up signal bit */
if(midiSignal!=-1)
{
    /* Set up a PlayerInfo with alarm clock */
    pPlayerInfo=CreatePlayer( PLAYER_Name,      "My_player",
                              PLAYER_Conductor, "My_conductor",
                              PLAYER_SignalTask, FindTask(NULL),
                              PLAYER_AlarmSigBit,midiSignal,
                              TAG_END);

    if(pPlayerInfo)
    {
        /* Start the realtime clock running */
        res = SetConductorState( pPlayerInfo, CLOCKSTATE_RUNNING, 0L );
        if(!res)
        {
            /* Set the realtime alarm clock */
            timerr = SetPlayerAttrs( pPlayerInfo,
                                    PLAYER_AlarmTime, 1000L,
                                    PLAYER_Ready,     TRUE,
                                    TAG_END);

            if(timerr)
            {
                /* You could do some other job before */
                /* calling Wait() on the alarm signal */
                Wait( 1L << midiSignal );

                else printf("Couldn't set alarm\n");
            }
            else printf("Couldn't start clock\n");

            DeletePlayer(pPlayerInfo);
        }
        else printf("Couldn't set up PlayerInfo structure.\n");

        FreeSignal(midiSignal);
    }
    else printf("Couldn't allocate signal.\n");
}

```

In the fragment above, the calling task requests a `PlayerInfo` with an alarm clock feature by passing the `PLAYER_AlarmSigBit` tag to `CreatePlayer()`. The `ti_Data` field of this tag contains the signal bit that will be set by the `realtime.library` when the alarm goes off. The `PLAYER_SignalTask` tag indicates which task will be signalled.

The realtime clock is then started by calling `SetConductorState()` (discussed below). This is important since any alarm requests made when the clock is stopped will be ignored.

Finally the alarm time is set by calling `SetPlayerAttrs()`. The parameters to this call are:

```

BOOL result = SetPlayerAttrs( struct PlayerInfo *pi, Tag tag, ...);

```

The `pi` parameter indicates which `PlayerInfo` structure is to have its attributes changed. The tag items indicate the attributes and their new values. If the change is made successfully, then `TRUE` is returned. `FALSE` indicates failure. In the fragment above, a wakeup time is requested using the `PLAYER_AlarmTime` tag. Also the calling task indicates to the Conductor that it is ready by using the `PLAYER_Ready` tag (more on this below).

At this point, the call to `Wait(1 << midiSignal)` will cause the calling task to sleep until time = 1000 ticks.

The discussion so far has concentrated on the alarm facility of the realtime library. An even finer level of control over time is available using the clock tick callback hook facility. Instead of setting an alarm to signal your task at some future time, the hook facilities allow your application code to be invoked *during every clock tick*.

To set up the callback hook, you use the `PLAYER_Hook` tag with the address of a standard Hook structure as defined in `<utilities/hook.h>`. This structure contains the address of the code you want to be invoked for every `realtime.library` clock tick. Here's a code fragment showing how this is done:

```
struct Task      *My_task;
LONG             My_signal,ticks;

...
ULONG __asm __interrupt __saves My_hookFunc (      /* Hook function set up, */
    register __a1 struct pmTime *msg,              /* structures and protos */
    register __a2 struct PlayerInfo *pi );          /* See the utility/hook.h */
                                                    /* header file and the */
                                                    /* Utility Library chapter */
                                                    /* of RCRM: Libraries for */
                                                    /* more information. */

struct Hook My_hook = {
    { NULL, NULL },
    My_hookFunc,
};

void main()
{
    struct PlayerInfo *pPlayerInfo=NULL;
    LONG res;

    ...Open the realtime.library and do other set up here...

    /* Create the player, install the hook and set up to Wait() 1000 ticks. */
    ticks=1000L;
    pPlayerInfo = CreatePlayer(PLAYER_Name,          "My_player",
                              PLAYER_Conductor,     "My_conductor",
                              PLAYER_Hook,           &My_hook,
                              PLAYER_UserData,      &ticks,
                              TAG_DONE );

    if (pPlayerInfo)
    {
        My_task = FindTask(NULL); /* Initialize these globals so that */
        My_signal=AllocSignal(-1L); /* the hook function can signal us. */
        if(My_signal!=-1L)

```

```

        {
            /* Start the clock running. */
            res=SetConductorState(pPlayerInfo,CLOCKSTATE_RUNNING,0L);
            if(!res)
            {
                ...your code goes here...
                Wait(1L<<My_signal | SIGBREAKF_CTRL_C);
            }
            FreeSignal(My_signal);
        }
        DeletePlayer(pPlayerInfo);
    }
    ...Close the library, etc....
}

```

In the code above the function named `My_hookFunc()` will be called by the `realtime.library` whenever a clock tick occurs. Register A1 will contain a pointer to a `pmTime` structure (defined in `<midi/realtime.h>`) and A2 will contain a pointer to the `PlayerInfo` which set up the callback hook.

Here's the callback hook function itself. This simply decrements a variable, `ticks`, whose address is pointed to by `pi->pi_UserData`. Notice how this address was filled in using the `PLAYER_UserData` tag in the call to `SetPlayerAttrs()` in `main()` above. When the variable, `ticks`, reaches zero, the hook function signals the main task.

```

ULONG __asm __interrupt __saves My_hookFunc ( /* Here is the hook function */
    register __a1 struct pmTime *msg,         /* that is called whenever the */
    register __a2 struct PlayerInfo *pi )      /* realtime.library clock */
{                                               /* ticks, i.e., 600 times per */
    switch (msg->pmt_Method)                   /* second. */
    {
        case PM_TICK:
            if ( (* (LONG *) (pi->pi_UserData)) > 0 ) /* This code simply decrements */
                (* (LONG *) (pi->pi_UserData))--;    /* a value whose address is */
            else                                     /* stored in the pi_UserData */
                Signal(My_task,1L << My_signal);     /* field of the PlayInfo that */
            break;                                  /* created this hook. When */
                                                    /* value reaches zero, the */
        default:                                    /* hook function signals the */
            break;                                  /* calling task. */
    }
    return 0L;
}

```

More About Conductors

It is the job of the Conductor to act as middle man between the Amiga's timing hardware and client tasks represented by `PlayerInfo` structures. Each Conductor keeps track of:

- ☐ an Exec list of all its client players
- ☐ the state of the conductor (i.e., running, stopped, paused, locating)
- ☐ what time it is relative to start time
- ☐ whether the Conductor is using the Amiga's internal CIAs for its timing pulses or an external source

As shown in the examples above, the "state" of the Conductor can be changed at any time by calling `SetConductorState()`. If the call succeeds, zero is returned:

```
LONG res= SetConductorState(struct PlayerInfo *pi, LONG newstate, LONG ti);
```

The `pi` parameter is a pointer to a `PlayerInfo` structure that is linked to the Conductor you want to change. The `ti` parameter is a time offset used for special cases (typically set to zero). The `newstate` parameter is one of the following:

CLOCKSTATE_STOPPED - The clock is not running

CLOCKSTATE_PAUSED - To the `realtime.library`, this is exactly the same as stopped. This is provided as a convenience to those applications that wish to make a distinction between the two.

CLOCKSTATE_RUNNING - The clock is running and time pulses are being distributed to any client applications (`PlayerInfos`) that are ready.

CLOCKSTATE_LOCATE - This is the same as running with one exception: the clock does not actually start until **all** client applications have indicated that they are ready by setting the `PLAYER_Ready` attribute of their `PlayerInfo` to `TRUE`. This allows applications that cannot start immediately to set up before timing pulses actually begin.

The difference between locating and running requires some explanation. Each `PlayerInfo` has a "ready bit" which is used to tell `RealTime` that the player is ready to receive ticks. This bit is reset each time that `RealTime` relocates the clock to a new time.

The reason for the ready bit is that some application need to find the appropriate location in the multimedia sequence or score before they can start playing at the new location. In some cases this can take a considerable amount of time. Hence, **CLOCKSTATE_LOCATE** is used with the ready bit to allow all players that are sharing a timing context to be synchronized together.

Players can set the state of their ready bit by using the `PLAYER_Ready` tag attribute either when the `PlayerInfo` is created with `CreatePlayer()` or later with `SetPlayerAttrs()`. See the first code fragment above for an example of this.

Further details of how the `realtime` library operates can be found in the "read me" files on the release disk. Be sure to read this for important information on how to maintain accuracy over long periods and tips on how to handle tempo for musically aware applications.

The MIDI System

The `camd.library` handles all the lower-level chores associated with handling MIDI data. These include dealing with serial port I/O at 31250 baud, coordinating the movement of MIDI data between applications that are running concurrently and routines for decoding and building MIDI messages.

The MIDI distribution system is based on the idea of *linkages* between applications. Each application or hardware driver can establish an I/O link to other applications or hardware drivers. The ability to have one application send data to another allows "pipelining" of the MIDI stream.

For instance, multiple links can be established to a single source, so that more than one application can see the same MIDI stream coming out of a hardware port or application output. Similarly, more than one application can send a MIDI stream to a single hardware port or application input.

Clusters, Links and Nodes

Applications are linked together at named rendezvous points known as clusters. (See the `MidiCluster` structure defined in the header file `<midi/camd.h>` for a listing.) There can be any number of `MidiCluster` structures and there is typically one for each hardware serial input and output port available.

There are normally two default clusters available in the system named "in.0" and "out.0" which correspond to the input and output hardware drivers for the Amiga's built-in serial port. To link up with these or any other `MidiClusters`, your application must create a `MidiNode` and a `MidiLink`.

The `MidiNode` structure is used as a central dispatch point for incoming and outgoing messages, and holds all of the application-specific information. Each application that wants access to MIDI data will have at least one `MidiNode` structure. These are created by calling `CreateMidi()`:

```
struct MidiNode *mn = CreateMidi( Tag tag, ... );
```

`CreateMidi()` takes a tag list specifying the attributes for the node as its parameter. (For a complete list of all tag attributes available, see the Autodoc for the `SetMidiAttrs()` function.) It returns a pointer to a `MidiNode` or `NULL` for failure.

Once the MidiNode is set up, it must be linked to a MidiCluster structure by setting up an appropriate MidiLink. To create the link call AddMidiLink():

```
struct MidiLink *ml = AddMidiLink(struct MidiNode *mn, LONG type, Tag tag, ...);
```

This function takes a pointer to the MidiNode to link, a type variable, either MLTYPE_Receiver or MLTYPE_Sender and a list of tags specifying the other attributes of the link. It returns a pointer to a MidiLink structure or NULL on failure.

Here's a fragment showing how to set up a MidiNode and link it with one of the default MidiClusters. (This code assumes you already have the camd.library open.)

```
pMidiNode *pMidiNode=NULL
pMidiLink *pMidiLink=NULL;

...

pMidiNode=CreateMidi( MIDI_Name,      "My_midiNode",
                     MIDI_MsgQueue, 0L,      /* This is a send-only */
                     MIDI_SysExSize,0L,      /* MIDI node so no input */
                     TAG_END);              /* buffers are needed. */

if(pMidiNode)
{
    pMidiLink=AddMidiLink( pMidiNode, MLTYPE_Sender,
                          MLINK_Comment, "My_midiLinkOut",
                          MLINK_Location, "out.0",
                          TAG_END);

    if(pMidiLink)
    {
        /* Your code goes here */

        RemoveMidiLink(pMidiLink);
    }
    DeleteMidi(pMidiNode);
}
```

In the code above, a MidiNode named "My_midiNode" is created and then an output link is established to "out.0", the default MIDI output hardware. This allows the calling application to merge their output with the output of any other application that has an output link to the same cluster. Note that before exiting, any MidiNodes should be deleted with DeleteMidi(); any MidiLinks should be removed with RemoveMidi().

One of the nice thing about CAMD's linking scheme is that links can be established even if the other application or hardware driver hasn't been loaded yet. This is because a MidiLink does not connect directly to the other application but to the MidiCluster or "named rendezvous point".

For example, if program-1 establishes an output link to the cluster "foo", and program-2 establishes an input link to that same cluster, then any data that program-1 sends to that link will be received by program-2. If a third application creates an input link to "foo", then it will

also receive the data, whereas if another application creates an output link to "foo" then its MIDI data will be merged with that of program-1, and distributed to all the input links.

So the rules of clusters are as follows:

- ☐ The first attempt to link to a cluster creates the cluster, and the last link to leave deletes it.
- ☐ Each sender link to a cluster is merged with all the other senders.
- ☐ Each receiver link to a cluster gets a copy of what all the other receivers get.

MIDI Messages

In CAMD, each MIDI message sent or received is contained in a MidiMsg structure.

```
typedef union
{
    ULONG l[2];
    UBYTE b[4];
} MidiMsg;
```

This 8-byte structure contains a timestamp, the actual MIDI bytes (up to 3) and a link number (so that applications which have several input links can determine which one sent the message). Note that since the message is so small, the entire message is copied when MIDI data is transferred, rather than passing pointers around.

How MIDI Data is Received

MIDI applications that receive data can be either task-based or callback based. A task-based application uses a signal to wait for incoming MIDI data. Once the signal is received, the application can call GetMidi() to actually look at what was received.

```
BOOL res = GetMidi(struct MidiNode *mn, MidiMsg *mm );
```

This function copies a message from the receive buffer of the MidiNode specified by mn to the MidiMsg structure specified by mm. If there were no messages in the buffer, FALSE is returned. TRUE indicates success.

All incoming messages are queued, and there is a separate queue for system exclusive messages (which can be quite long). Each incoming MIDI event is both timestamped and marked with the linkage number (settable by the application) from the link that it came in on.

Task based applications should have their priority set high enough to avoid missing bytes. A high-priority task (say, 30 or so), can meet professional requirements, even when the disk drive is running.

However, if the application's handling of MIDI is very fast, it may be better to use a callback. The callback occurs in the context of the sender, so it is best to be quick so as not to slow down the sending task. (Note that the sender will always be a task, and not an interrupt, since the actual hardware drivers are serviced via a task) The callback is invoked through a standard Hook structure. Using a callback avoids the overhead of task switching, and can allow improved overall performance.

How MIDI Data is Sent

Sending MIDI data is very simple, mainly a matter of filling out a MidiMsg structure and calling PutMidi().

```
void PutMidi(struct MidiLink *ml, ULONG Msg);
```

Note that if the receiver's buffer is full, then the function will fail, rather than waiting for the receive buffer to empty. Another way of sending MIDI data is available with the ParseMidi() function:

```
void ParseMidi(struct MidiLink *ml, UBYTE *buf, ULONG len);
```

This function allows you to send a buffer full of MIDI bytes to the MidiLink specified by ml. The buffer is pointed to by buf and its length is specified by len. The MidiLink should be MLTYPE_Sender.

Standard Midi File Player

Listed below is a simple MIDI file player which allows the playback of standard MIDI files from the Amiga's Shell interface. This program demonstrates many of the features of the realtime.library and camd.library discussed in this article.

To use it simply enter the program name,smf, followed by the name of a standard MIDI file. Note that this program is not sophisticated in its handling of tempo and should not be used as a model for musically-aware applications.

There are other important sources of information that you should consult for further details about CAMD:

- ☐ The realtime and camd header files
- ☐ The realtime and camd Autodocs
- ☐ The examples on the release disk
- ☐ The read-me files, camd.readme, ecamd.readme and realtime.readme on the release disk

Standard MIDI File Player

```

// Execute this file to compile it under SAS/Lattice C 5.10b
-i1 -cand:include -cfirst -v -j73 smf.c
link FROM LIB:c.o smf.o to smf LIB:ic.lib LIB:amiga.lib
quit
//

```

This file is a prototype score player which reads standard MIDI files, formats them to type 0 or 1. It conforms to the 1.0 specification for standard MIDI files by Dave Oppeheim of Opcode systems. It uses both the CMD and REALTIME libraries by Roger Dannenberg, David Joiner, et al. Run from the shell/CLI only. Usage: mmf <standard MIDI file name>

Design and coding by Dan Baker, Commodore Business Machines.

The tempo handling used by this program is crude and is known to have deficiencies. It should not be used as an example of proper tempo handling. This program also makes extensive use of globals and other techniques which are convenient to the program's author. Apologies to the reader.

```

/* System Include Files */
#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dosextens.h>

/* CMD MIDI Library Includes */
#include <clib/camd_protos.h>
#include <aidi/camd.h>
#include <aidi/candbase.h>
#include <pragmas/camd_pragmas.h>

/* CMD Real Time Library Includes */
#include <clib/realtime_protos.h>
#include <aidi/realtime.h>
#include <aidi/realtimebase.h>
#include <pragmas/realtime_pragmas.h>

/* System function prototypes */
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

/* Lattice Standard I/O */
#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CHRM(void) { return(0); }
int chabort(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

/*-----*/
/* Prototypes */
void kill(char *killstring);
ULONG ComLen (UBYTE *value);
UBYTE *DecodeEvent (UBYTE *, struct SMFTrack *, ULONG);
ULONG transfer(struct SMFTrack *, ULONG, ULONG);
ULONG changetempo (ULONG);

/*-----*/
/* S M F Header File */
/*-----*/
/* Four-character Identifier builder*/
#define MakeID(a,b,c,d) ((LONG)(a)<<24L | (LONG)(b)<<16L | (c)<<8 | (d))
#define Mthd MakeID('M','T','h','d')
#define Mtrk MakeID('M','T','r','k')

```

```

struct SMFHeader {LONG
  ChunkID; /* 4 ASCII characters */
  VarLang;
  Format;
  WORD
  WORD
  WORD
  WORD
  Division;
};

struct DecTrack {
  ULONG absdelta; /* 32-bit delta */
  ULONG nresclock; /* storage */
  BYTE status; /* status from file */
  BYTE rstatus; /* running status from track */
  BYTE d1; /* data byte 1 */
  BYTE d2; /* data byte 2 */
  ULONG absamlength; /* 32-bit absolute metalength */
  BYTE *endmarker; /* meta event type */
  BYTE metatype; /* meta event type */
  BOOL playable;
  BYTE pad3;
};

/*-----*/
/* MIDI Header File */
/*-----*/
#define MAXTRAX 16L
#define TWO_MAXTRAX 32L
#define MIDIBUFSIZE 512L

/* Library Base */
struct Library {
  CmdBase;
  RealTimeBase;
};

/* Compiler glue: stub functions for cmd.library */
struct MidiNode {
  struct MidiNode *CreateMidi(Tag tag, ...)
  {
    return CreateMidi((struct TagItem *)tag);
  }
};

/* SetMidiAttrs(struct MidiNode *mi, Tag tag, ...)
{
  return SetMidiAttrs(mi, (struct TagItem *)tag);
}

/* AddMidiLink(struct MidiNode *mi, LONG type, Tag tag, ...)
{
  return AddMidiLink(mi, type, (struct TagItem *)tag);
}

/* Compiler glue: stub functions for realtime.library */
struct SetPlayerAttrs(struct PlayerInfo *pi, Tag tag, ...)
{
  return SetPlayerAttrs(pi, (struct TagItem *)tag);
}

/* CreatePlayer(Tag tag, ...)
{
  return CreatePlayer((struct TagItem *)tag);
}

/*-----*/
/* Globals */
/*-----*/

struct MidiLink {
  struct MidiNode *pmidilink;
  struct MidiNode *pmidinode;
  struct PlayerInfo *pplayerinfo;
  SMFTR
  BYTE
  BYTE
  ULONG
  ULONG
  *oldclock, *sizeoftrack, *tfactor, *division, *donecount;
  *pfillbuf[2], *lastRSCChan;
};

```

```

/*-----*/
/* CODE Starts Here */
/*-----*/
ULONG
ComVarLen (value)
BYTE *value;
{
    register ULONG newval;
    register BYTE x;
    x=0;newval=0L;
    while (x<4)
    {
        newval <= 7;
        newval |= *(value+x) & 0x7ff;
        if (*(value+x) < 0x80) x=4;
        x++;
    }
    return(newval);
}

void
main(int argc, char **argv)
{
    char *smfname, iobuffer[14];
    struct SMFHeader
    {
        BYTE *pSMFHeader;
        WORD w;
        LONG rdcount, y, z, res;
        BOOL notdone, timer;
        ULONG masterswitch, lowclock,
        ylength[2], wakeup;
    };
    BYTE oldpri; /* Priority to restore */
    struct Task
    {
        BYTE fillbuf1[MIDIMUPSIZE]; /* These buffers hold the notes translated */
        BYTE fillbuf2[MIDIMUPSIZE]; /* from the SMF file for playback */
        struct DecTrack *pTrack[MAXTRAX];
        oldclock=0L;
        smfdata=NULL;
        smfhandle=0;
        pidiLink=NULL;
        pidiMode=NULL;
        pPlayerInfo=NULL;
        RealTimeBase=NULL;
        CamdBase=NULL;
        notdone=TRUE;
        fillbuf[0]=fillbuf1;
        fillbuf[1]=fillbuf2;
        fillclock[0]=0L;
        fillclock[1]=0L;
        donecount=0L;
        last8char=0x0f; /* Status of $f1 is undefined in standard MIDI file spec */
        tfactor=12;
    };
    /*-----*/
    /* Open the CMD and RealTime libraries */
    /*-----*/
    CamdBase=OpenLibrary("camd.library",0L);
    if (!CamdBase)
        kill("Can't open CMD MIDI Library\n");
    RealTimeBase=OpenLibrary("realtime.library",0L);
    if (!RealTimeBase)
        kill("Can't open CMD Real Time Library\n");
}

```

```

/*-----*/
/* Open the File */
/*-----*/
if (argv[1]==NULL)
    kill("No file name given.\n");
smfname=argv[1];
smfhandle=Open( (BYTE *)smfname , MODE_OLDFILE );
if (smfhandle==0)
    kill("Cannot open file.\n");

/*-----*/
/* Read the SMF Header ID/Length */
/*-----*/
rdcount = Read(smfhandle,iobuffer,14);
if (rdcount==1)
    kill ("Bad file during read.\n");
if (rdcount<14)
    kill ("Not an SMF file, too short.\n");

/*-----*/
/* Evaluate Header */
/*-----*/
pSMFHeader=(struct SMFHeader *)iobuffer;
if (pSMFHeader->ChunkID != MThd )
    kill("SMF file has unknown header ID.\n");
if (pSMFHeader->VarLeng != 6 )
    kill("SMF file has unknown header.\n");
if (pSMFHeader->Format == 0 )
    printf("Parsing SMF file format type 0\n");
else if (pSMFHeader->Format == 1 )
    printf("Parsing SMF file format type 1\n");
else
    kill("Can't parse this SMF file type.\n");

if (pSMFHeader->Nrtrks > MAXTRAX )
    kill("SMF file has more than MAXTRAX tracks.\n");
else
    printf("SMF file has %ld tracks\n",pSMFHeader->Nrtrks);

/*-----*/
/* Evaluate time base */
/*-----*/
if (pSMFHeader->Division < 0)
{
    /* Real time; find frame rate */
    w=(pSMFHeader->Division > 8) & 0x007f;
    if (w==24 && w!=25 && w!=30)
        kill("Non-metrical time specified; not MIDI/SMF frame rate\n");
    else
        kill("Non-metrical time; MIDI/SMF frame rate\n");
}
/* Real-time; find seconds resolution */
w=(pSMFHeader->Division & 0x007f);
if (w==4)
    kill("Non-metrical time in quarter seconds (MTC resolution)\n");
else if (w==8 || w==10 || w==80)
    kill("Non-metrical time in 1/nths of a second (bit resolution)\n");
else
    kill("Non-metrical time in 1/nths of a second\n");
}
else
{
    printf("SMF file has 1/width quarter notes per delta tick\n",
        pSMFHeader->Division);
    division=(ULONG)pSMFHeader->Division;
}
}

```



```

/*-----*/
/* Transfer second events to B buffer */
/*-----*/
y=0;
masterswitch=1L;
lowclock=0xffffffff;
for (x=0;x<trackct;x++)
{
    if (pDTrack[x]->nextclock < lowclock && pTrack[x])
        lowclock=pDTrack[x]->nextclock;
}
for (x=0;x<trackct;x++)
{
    if (pDTrack[x]->nextclock==lowclock && pTrack[x])
    {
        /* Transfer event to parse buffer and handle successor */
        y=transfer(pDTrack[x],masterswitch,y);
        x++;
        while (z==1)
        {
            pTrack[x]=DecodeEvent(pTrack[x],pDTrack[x],masterswitch);
            /* Next delta is zero... */
            if ( ! (pDTrack[x]->abedelta) && pTrack[x])
            {
                y=transfer(pDTrack[x],masterswitch,y);
            }
            else (z=0);
        }
    }
}
ylength[masterswitch]=y;
fillclock[masterswitch]= (LONG) (tfactor*lowclock);

/*-----*/
/* Priority Must Be Above Intuition and Graphics Stuff */
/*-----*/
m=FindTask(NULL);
oldpri=SetTaskPri(m,21);

/*-----*/
/* Set up a PlayerInfo and a Conductor to get timing information */
/*-----*/
midSignal = AllocateSignal(-1L);
if (midSignal==1)
    kill("Couldn't allocate a signal bit\n");

playerInfo=CreatePlayer( PLAYER_Name, "SNP Player PlayerInfo",
                        PLAYER_Conductor, "SNP Player's Conductor",
                        PLAYER_SignalTask, m,
                        PLAYER_AlarmSignal, midSignal,
                        TAG_END);

if (!pPlayerInfo)
    kill("Can't create a PlayerInfo\n");

/*-----*/
/* Make sure the clock is stopped. */
/*-----*/
res = SetConductorState( pPlayerInfo, CLOCKSTATE_STOPPED, 0L );
if (res!=0)
    kill("Couldn't stop conductor\n");

/*-----*/
/* Play the first batch of notes in Buffer A */
/*-----*/
if (ylength[masterswitch^1] != 0)
{
    ParseMid(pMidLink, pFillbuf[masterswitch^1],
            ylength[masterswitch^1]);
}

```

```

/*-----*/
/* and start the Realtime alarm clock */
/*-----*/
res = SetConductorState( pPlayerInfo, CLOCKSTATE_RUNNING, 0L );
if (res!=0)
    kill("Couldn't start conductor\n");

/*-----*/
/* and set the alarm. */
/*-----*/
timerr = SetPlayerAttr( pPlayerInfo,
                        PLAYER_AlarmTime, fillclock[masterswitch],
                        PLAYER_Ready, TRUE,
                        TAG_END);

if (!timerr)
    kill("Couldn't set player attr\n");

/*-----*/
/* MAIN EVENT LOOP */
/*-----*/
while (donecount<trackct)
{
    masterswitch ^= 1L;
    y=0;
    lowclock=0xffffffff;

    /* Get events from track into DecTrack structures */
    for (x=0;x<trackct;x++)
    {
        if (pDTrack[x]->nextclock < lowclock) && pTrack[x])
            lowclock=pDTrack[x]->nextclock;
    }

    /* Transfer events to current buffer */
    for (x=0;x<trackct;x++)
    {
        if (pDTrack[x]->nextclock==lowclock) && pTrack[x])
        {
            /* Transfer event to parse buffer and handle successor */
            y=transfer(pDTrack[x],masterswitch,y);
            z=1;
            while (z==1)
            {
                pTrack[x]=DecodeEvent(pTrack[x],pDTrack[x],masterswitch);
                /* Next delta is zero... */
                if ( ! (pDTrack[x]->abedelta) && pTrack[x])
                {
                    y=transfer(pDTrack[x],masterswitch,y);
                }
                else (z=0);
            }
        }
        ylength[masterswitch]=y;
        fillclock[masterswitch]= (LONG) (tfactor*lowclock);

        /* Wait() for the CMD alarm or a CTRL-C keypress from the user. */
        /*-----*/
        if (timerr)
            wakeup=Wait(1L<< midSignal | SIGBREAKF_CTRL_C);
    }
}

```

```

if (wakeUp & SIGBREAKF_CTRL_C)
{
    /* Restore Priority */
    if (nt != NULL) SetTaskPri (nt, oldpri);
    /* And Quit */
    kill ("User abort\n");
}

/*-----*/
/* Start the next set of events */
/*-----*/
if (ylength[masterSwitch-1] != 0)
{
    ParamMidi (pMidiLink, pfillbuf[masterSwitch-1],
              ylength[masterSwitch-1]);
}

/*-----*/
/* and set the alarm. */
/*-----*/
timerr = SetPlayerAttrs (pPlayerInfo,
                        PLAYER_AlarmTime, fillclock(masterSwitch),
                        PLAYER_Ready, TRUE,
                        TAG_END);

if (!timerr)
    kill ("Couldn't set player attrs 2\n");
}

/*-----*/
/* Finish off the last set of events */
/*-----*/
masterSwitch = 1;

if (timerr)
    wakeupWait (1L << midisignal | SIGBREAKF_CTRL_C);

if (ylength[masterSwitch-1] != 0)
    ParamMidi (pMidiLink, pfillbuf[masterSwitch-1],
              ylength[masterSwitch-1]);
}

/* Restore Priority */
if (nt != NULL) SetTaskPri (nt, oldpri);

kill ("NOX\n");
}

/*-----*/
/* Cleanup and return resources */
/*-----*/
void
kill (char *killstring)
{
    if (midisignal != -1) FreeSignal (midisignal);
    if (pPlayerInfo) DeletePlayer (pPlayerInfo);
    if (pData) FreeMem (pData, sizeof (pData));
    if (pMidiLink) RemoveMidiLink (pMidiLink);
    if (pMidiNode) DeleteMidi (pMidiNode);
    if (RealTimeBase) CloseLibrary (RealTimeBase);
    if (CamBase) CloseLibrary (CamBase);
    if (smfhandle) Close (smfhandle);
    if (smfdata) FreeMem (smfdata, smfdatasize);
    printf (killstring);
    exit (0);
}

```

```

/*-----*/
/* Translate from raw track data to a decoded event */
/*-----*/
UNYTE *DecodeEvent (UNYTE *pdata, struct DecTrack *pOTdata, ULONG deaswitch)
{
    LONG status;
    ULONG length;
    BOOL skipit;

    pOTdata->absdelta = 0;
    pOTdata->playable = TRUE; /* Assume it's playable and not a meta-event */
    do
    {
        skipit = FALSE;
        /* Is this track all used up? */
        if (pdata >= pOTdata->endmarker)
        {
            printf ("Track done...\n");
            donecount++;
            return (0);
        }
        else /* there is more data to handle in the track */
        {
            /* Decode delta */
            pOTdata->absdelta += ComVarLen (pdata);
            pOTdata->nextclock += pOTdata->absdelta;
            /* Update pointer to event following delta */
            while ("pdata" > 127)
            {
                pdata++;
            }
            pOTdata++;
        }
        if ("pdata" > 127) /* Event with status ($80-$FF): decode new status */
        {
            status = *pdata;
            pOTdata->status = status; /* No running status was used */
            pOTdata++;
        }
        if (status < 240) /* Handle easy status $8x - $Ex */
        {
            skipit = FALSE;
            pOTdata->d1 = *pdata;
            if (status < 192 || status > 223) /* $80-$8F, $E0-$EF: 2 data bytes */
            {
                pdata++;
                pOTdata->d2 = *pdata;
            }
            else pOTdata->d2 = 0; /* $C0-$DF: 1 data byte */
        }
        else /* Status byte $Fx, system exclusive or meta events */
        {
            skipit = TRUE;
            if (status == 0xFF) /* It's a meta event ($ff) */
            {
                pOTdata->metatype = *pdata;
                pdata++; /* Now pointing at length byte */
                if (pOTdata->metatype == 81)
                {
                    /* Tempo change event. There are 60 million
                     * microseconds in a minute. The lower 3 bytes
                     * pointed to by pdata give the microseconds
                     * per MIDI quarter note. So, assuming a quarter

```

```

/* note gets the beat, this equation
*/
/* 60000000L /
*/
/* ((ULONG *)pdata) & 0x00ffffff ) */
/* gives beats per minute.
*/
tfactor = chngtempo( *((ULONG *)pdata) & 0x00ffffff );

/* Tempo event is not playable. This prevents the
*/
/* event from being transferred to the play buffer */
pdata->playable = FALSE;

/* Even though this event can't be played, it
*/
/* takes some time and should not be skipped.
*/
skipit=FALSE;
length=ConvVarLen(pdata);
pdata->abamlength=length;
while(*pdata>12) pdata++;
pdata+=length;
}
else if(status==0xf0 || status==0xf7) /* It's a sysex event */
{
    pdata->metatype=0xff;
    printf("sysex event");
    length=ConvVarLen(pdata);
    pdata->abamlength=length;
    while(*pdata>12) pdata++;
    pdata+=length;
}
else /* It's an unknown event type ($f1-$fe, $fa-$fe) */
{
    pdata->metatype=0xff;
    printf("Unknown event");
}
}
else /* Event without status ($00-$7f): use running status */
{
    skipit=FALSE;
    /* Running status data bytes */
    status=pdata->status;
    pdata->rstatus=status;
    if(status==0)
        kill("Bad file, data bytes without initial status.\n");
    pdata->d1=*pdata;
    if(status<192 || status>223) /* $80-$8f, $80-$8f: 2 data bytes */
    {
        pdata++;
        pdata->d2=*pdata;
    }
    else pdata->d2=0; /* $c0-$df: 1 data byte */
    pdata++;
    while(skipit);
    return(pdata);
}
}

```

```

/* Transfer the decoded event to the fill buffer for playback */
/*
LONG
transfer(struct DecTrack *pDT, ULONG mawitch, LONG ylen)
{
    ULONG y;
    y=(ULONG) ylen;
    if (pDT->playable)
    {
        if(pDT->rstatus == lastRChan)
        {
            /* Running status so just put the 2 data bytes in buffer */
            *(pfillbuf[mawitch] + y)-pDT->d1;
            y++;
            *(pfillbuf[mawitch] + y)-pDT->d2;
        }
        else
        {
            /* New status so store status and data bytes */
            *(pfillbuf[mawitch] + y)-pDT->status;
            y++;
            *(pfillbuf[mawitch] + y)-pDT->d1;
            if(pDT->status<192 || pDT->status>223)
            {
                y++;
                *(pfillbuf[mawitch] + y)-pDT->d2;
            }
            lastRChan=pDT->status;
        }
        y++;
    }
    return((LONG)y);
}

/* Handle the Change Tempo event. With the realtime.library, the timing
*/
/* quantum is fixed at 1.66 milliseconds (600 Hz). This makes handling
*/
/* of SMF tempo pretty rough. Tempo is controlled through a ULONG integer
*/
/* named tfactor which is used to multiply the time deltas in the SMF
*/
/* file. We can't multiply the time deltas by a fractional amount and
*/
/* that makes tempo handling even rougher. For correct tempo handling the
*/
/* clock quantum has to be variable and controlled by this program.
*/
ULONG
chngtempo(ULONG ctbpm)
{
    ULONG timefac, timerem, tickfreq;
    tickfreq=(ULONG)((struct RealTimeBase *)RealTimeBase->TickFreq);
    /* CMD uses 1.66ms quantum for 600 ticks/sec
    */
    /* SMF uses one tick = quarter note/pMFHeader->Division
    */
    /* Hence, microseconds per delta in SMF is given by ctbpm/division
    */
    /* and BM is given by 60,000,000/ctbpm.
    */
    timefac=(tickfreq * (ctbpm/division)) / 1000000;
    timerem=(tickfreq * (ctbpm/division)) % 1000000;
    if ( timerem >= 500000 )
        timefac++;
    return(timefac);
}

```


Realtime Library Autodocs

TABLE OF CONTENTS

realtime.library/CreatePlayer
 realtime.library/DeletePlayer
 realtime.library/ExternalSync
 realtime.library/FindConductor
 realtime.library/GetPlayerAttrA
 realtime.library/LockRealtime
 realtime.library/NextConductor
 realtime.library/SetConductorState
 realtime.library/SetPlayerAttrA
 realtime.library/UnlockRealtime

realtime.library/CreatePlayer realtime.library/CreatePlayer
 NAME CreatePlayer -- Create a PlayerInfo and link to a Conductor.
 SYNOPSIS
 struct PlayerInfo *CreatePlayerA (struct TagItem *TagList)
 struct PlayerInfo *CreatePlayer (Tag tag1, ...)
 FUNCTION
 Creates a PlayerInfo structure with the desired attributes.
 The first form of the function expects a tag array pointer or NULL.
 The second form permits the tag items to exist on the caller's
 stack. In both cases, the final tag item must be TAG_END.
 INPUTS
 TagList - optional pointer to tag array. May be NULL. For
 OS v1.3, there are restrictions on the tag array
 contents. See NOTE below.
 TAGS See SetPlayerAttrA()
 RESULTS
 A pointer to a PlayerInfo structure on success or NULL on failure.
 When NULL is returned, an error code will be returned if a TagItem
 with an ti_Tag of PLAYER_ErrorCode was provided. The RTE_error code
 will be put in the ti_Data field.
 NOTE Under 1.3 only a restricted tag array may be passed into this
 function. The only special tag values that are supported are those
 documented for Release 2.04. The full range of TAG list operations
 are permitted under 2.0.
 EXAMPLES
 pl = CreatePlayer (
 PLAYER_Conductor, "metric", PLAYER_Priority, 10, TAG_END);
 SEE ALSO
 DeletePlayer(), SetPlayerAttrA()
 2.0 tag docs.

realtime.library/DeletePlayer realtime.library/DeletePlayer

NAME DeletePlayer -- Delete a PlayerInfo structure

SYNOPSIS
void DeletePlayer (struct PlayerInfo *pi)

FUNCTION
Deletes the specified PlayerInfo. The following actions are automatically performed by this function:
Flushes the Conductor that the PlayerInfo was connected to if this is the last PlayerInfo connected to that Conductor.

INPUTS
pi - PlayerInfo to delete. Can be NULL.

RESULTS
None

SEE ALSO
CreatePlayer()

realtime.library/ExternalSync realtime.library/ExternalSync

NAME ExternalSync -- External source clock for a PlayerInfo's Conductor

SYNOPSIS
BOOL ExternalSync (struct PlayerInfo *pi, LONG mintime, LONG maxtime)

FUNCTION
Allows external application to constrain conductor time between mintime and maxtime, with the restraint that time can never run backwards. Does nothing if the given PlayerInfo is not the current External Sync source (as indicated by the PLAYERS_EXTSYNC flag).

INPUTS
pi - PlayerInfo referencing the Conductor to change
mintime, maxtime - time constraints

RESULTS
result - TRUE if everything went OK, FALSE if not the current sync source or no conductor


```

realtime.library/LockRealTime      realtime.library/LockRealTime

NAME
    LockRealTime -- Prevent other tasks from changing internal structures

SYNOPSIS
    LockRealTime(locktype)
    do
    APTR LockRealTime( ULONG );

FUNCTION
    This routine will lock the internal semaphores in the RealTime library.
    If they are already locked by another task, this routine will wait
    until they are free.

INPUTS
    locktype -- which internal list will be locked.
    RT_Conductors -- locks the internal list of Conductors and
    associated structures.

RESULT
    If locktype is valid, returns a value that must be passed later
    to UnlockRealTime.

EXAMPLE
NOTES
BUGS
SEE ALSO
    UnlockRealTime()

```

```

realtime.library/NextConductor      realtime.library/NextConductor

NAME
    NextConductor -- Return next Conductor on RealTime Conductor list

SYNOPSIS
    struct Conductor *NextConductor (struct Conductor *last)
    do

FUNCTION
    Returns the next Conductor on RealTime Conductor list. If last is NULL,
    returns the first Conductor. Returns NULL if no more Conductors.
    Conductor links (RT_Conductors) must be locked when called.

INPUTS
    last - previous Conductor or NULL to get first Conductor

RESULTS
    next - next Conductor or NULL

```

```

realtime.library/SetConductorState      realtime.library/SetConductorState

NAME      SetConductorState -- Change the play state of a PlayerInfo's conductor

SYNOPSIS  LONG SetConductorState (struct PlayerInfo *pi, LONG state, LONG time)
          a0                                d0                                d1

FUNCTION
  Changes the play state of the Conductor referenced by the
  PlayerInfo. The states can be CLOCKSTATE_STOPPED, CLOCKSTATE_PAUSED,
  CLOCKSTATE_LOCATE, CLOCKSTATE_RUNNING, or the special value
  CLOCKSTATE_METRIC which asks the highest priority conducted node
  to do a CLOCKSTATE_LOCATE, or the special value CLOCKSTATE_SHUTTLE
  which informs the players that the clock value is changing, but the
  clock isn't actually running. Note that going from CLOCKSTATE_PAUSED
  to CLOCKSTATE_RUNNING does not reset the ClockTime of the Conductor.

INPUTS
  pi - PlayerInfo referencing the Conductor to change
  state - new play state of Conductor
  time - start time offset in RealTime heartbeat units

RESULTS
  result - 0 if everything went OK, else an error code

```

```

realtime.library/SetPlayerAttrs      realtime.library/SetPlayerAttrs

NAME      SetPlayerAttrs -- Set the attributes of a PlayerInfo

SYNOPSIS  BOOL SetPlayerAttrsA (struct PlayerInfo *pi, struct TagItem *tags)
          a0                                a1

FUNCTION
  Sets the attributes of a PlayerInfo, using a Tag List.

  The first form of the function expects a tag array pointer or NULL.
  The second form permits the tag items to exist on the caller's
  stack. In both cases, the final tag item must be TAG_END.

INPUTS
  pi - a pointer to the PlayerInfo structure

  TagList - optional pointer to tag array. May be NULL. For
  OS v1.3, there are restrictions on the tag array
  contents. See NOTE below.

TAGS
  PLAYER_Name      STRPTR - ti Data points to the new name of
                    the PlayerInfo (generally the Application name)
  PLAYER_Hook      struct Hook * - this hook will be called when
                    time changes occur.
  PLAYER_Priority  BYTE - new priority for the PlayerInfo
  PLAYER_Conductor STRPTR - ti Data points to the name of the
                    Conductor to link with. If NULL, delink from conductor.
                    If -0, create private conductor.
  PLAYER_Ready     BOOL - set/clear the "ready" flag
  PLAYER_SignalTask struct Task * - task to signal on notify or alarm
  PLAYER_AlarmSigBit BYTE - signal bit to use on alarm or
                    -1 to disable
  PLAYER_Quiet     BOOL - when TRUE, this node is ignored. Mainly
                    used by external sync applications
  PLAYER_UserData  void * - sets UserData field
  PLAYER_ID        UWORD - sets PlayerID field
  PLAYER_AlarmTime LONG - sets the AlarmTime and the ALARMSET flag
  PLAYER_AlarmOn    BOOL - if TRUE sets the ALARMSET flag, FALSE clears
                    the flag
  PLAYER_Conducted BOOL - if TRUE sets the CONDUCTED flag, FALSE clears
                    the flag
  PLAYER_ExtSync    BOOL - attempt to become external sync source (TRUE)
                    or release external sync (FALSE)

RESULTS
  TRUE if all changes were made successfully or FALSE on failure.
  When FALSE is returned, an error code will be returned if a TagItem
  with an ti_Tag of PLAYER_ErrorCode was provided. The RTE_error code
  will be put in the ti_Data field.

NOTE
  Under 1.3 only a restricted tag array may be passed into this

```

function. The only special tag values that are supported are those documented for Release 2.04. The full range of TAG list operations are permitted under 2.0.

EXAMPLES

```
result = SetPlayerAttrs (pl,
    PLAYER_Conductor, "metric", PLAYER_Priority, 10, TAG_END);
```

Modifies a PlayerInfo to so it links to the Conductor named "metric" and sets the priority of the PlayerInfo to +10.

SEE ALSO

CreatePlayer(), DeletePlayer()
2.0 tag docs.

realtime.library/UnlockRealTime realtime.library/UnlockRealTime

NAME

UnlockRealTime -- Unlock internal lists

SYNOPSIS

```
UnlockRealTime(lock)
```

and

```
void UnlockRealTime( APTR );
```

FUNCTION

Undoes the effects of LockRealTime().

INPUTS

lock -- value returned by LockRealTime(). Can be NULL.

EXAMPLE

SEE ALSO

LockRealTime()

CAMD Library Autodocs

TABLE OF CONTENTS

camd.library/AddMidLink
 camd.library/CloseMidDevice
 camd.library/CreateMidId
 camd.library/DeleteMidId
 camd.library/EndClusterNotify
 camd.library/FindMidId
 camd.library/FlushMidId
 camd.library/GetMidId
 camd.library/GetMidIdAttrA
 camd.library/GetMidIdErr
 camd.library/GetMidLinkAttrA
 camd.library/GetSysEx
 camd.library/LinkCMD
 camd.library/MidLinkConnected
 camd.library/MidLinkLen
 camd.library/MidLinkType
 camd.library/NextCluster
 camd.library/NextClusterLink
 camd.library/NextMidId
 camd.library/NextMidLink
 camd.library/OpenMidDevice
 camd.library/ParseMidId
 camd.library/PutMidId
 camd.library/PutMidMsg
 camd.library/PutSysEx
 camd.library/QuerySysEx
 camd.library/RemoveMidLink
 camd.library/ReThinkCMD
 camd.library/SetMidIdAttrA
 camd.library/SetMidLinkAttrA
 camd.library/SkipSysEx
 camd.library/StartClusterNotify
 camd.library/UnlockCMD
 camd.library/WaitMidId

camd.library/AddMidLink

NAME

AddMidLink -- Create a MidLink to a MidCluster.

SYNOPSIS

```

struct MidLink *AddMidLinkA (struct MidNode *mi, LONG type,
                             a0
                             struct TagItem *TagList)
                             a1
                             Tag type1, ...)

```

FUNCTION

Creates a MidLink structure and connects it to a MidCluster.

The first form of the function expects a tag array pointer or NULL. The second form permits the tag items to exist on the caller's stack. In both cases, the final tag item must be TAG_END.

INPUTS

mi - MidNode that the MidLink should communicate through.

type - Type of link, either MLTYPE_Receiver or MLTYPE_Sender.

TagList - optional pointer to tag array. May be NULL. For OS v1.3, there are restrictions on the tag array contents. See NOTE below.

TAGS

See SetMidLinkAttrA

RESULTS

A pointer to a MidLink structure on success or NULL on failure. When FALSE is returned, an error code will be returned if a TagItem with an ti_Tag of MLINK_ErrorCode was provided. The CME_error Code will be put in the ti_Data field.

NOTE

Under 1.3 only a restricted tag array may be passed into this function. Specifically, the only special tag values that are supported are TAG_END (or TAG_DONE) and TAG_IGNORE. All others (e.g. TAG_SKIP, TAG_MORE) are treated as TAG_END. The full range of TAG list operations are permitted under 2.0.

This function MUST be called by a Process (not a Task). It makes a few DOS calls.

EXAMPLES

```

mi = AddMidLink (mi, MLTYPE_Receiver,
                 MLINK_Name, "app.in", MLINK_Location, "in.0", TAG_END);

```

Creates a receive MidLink named "app.in" to the MidCluster named "in.0", where all bytes are received through the MidNode mi.

SEE ALSO

RemoveMidLinkA(), SetMidLinkAttrA()
2.0 tag docs.

camd.library/CloseMidiDevice camd.library/CloseMidiDevice

NAME CloseMidiDevice -- Close a MIDI device driver.

SYNOPSIS

```
void CloseMidiDevice (struct MidiDeviceData *MidiDeviceData)
                        ^0
```

FUNCTION

 Closes a MIDI device driver opened by OpenMidiDevice().

INPUTS

 MidiDeviceData - pointer to a MidiDeviceData structure returned by OpenMidiDevice().

RESULTS

 None

SEE ALSO

 OpenMidiDevice()

camd.library/CreateMidi camd.library/CreateMidi

NAME CreateMidi -- Create a MidiNode.

SYNOPSIS

```
struct MidiNode *CreateMidi (struct TagItem *TagList)
                        ^0
```

 struct MidiNode *CreateMidi (Tag tag1, ...)

FUNCTION

 Creates a MidiNode structure with the desired attributes.

 The first form of the function expects a tag array pointer or NULL. The second form permits the tag items to exist on the caller's stack. In both cases, the final tag item must be TAG_END.

INPUTS

 TagList - optional pointer to tag array. May be NULL. For OS v1.3, there are restrictions on the tag array contents. See NOTE below.

TAGS

 See SetMidiAttrs

RESULTS

 A pointer to a MidiNode structure on success or NULL on failure. When FALSE is returned, an error code will be returned if a TagItem with an ti_Tag of MIDI_ErrorCode was provided. The CMS_error code will be put in the ti_Data field.

NOTE

 Under 1.3 only a restricted tag array may be passed into this function. Specifically, the only special tag values that are supported are TAG_END (or TAG_DONE) and TAG_IGNORE. All others (e.g. TAG_SKIP, TAG_MORE) are treated as TAG_END. The full range of TAG list operations are permitted under 2.0.

 This function MUST be called by a Process (not a Task). It makes a few DOS calls.

 Don't call WaitMidi() with a send-only MidiNode.

EXAMPLES

```
mi = CreateMidi (
    MIDI_MagQueue, 2048, MIDI_SysExSize, 10000L, TAG_END) ;
```

 Creates a MidiNode with space to receive 2048 MidiMsg's and 10000 bytes of SysEx data. Note the 'L' on the end of 10000. This forces correct alignment of the tag items on the stack for 16 bit integer compilation.

SEE ALSO

 DeleteMidi(), SetMidiAttrs()
 2.0 tag docs.


```

camd.library/DeleteMidi                                camd.library/DeleteMidi

NAME      DeleteMidi -- Delete a MidiNode.
SYNOPSIS  void DeleteMidi (struct MidiNode *mn)
          a0
FUNCTION  Delete the specified MidiNode. A sys/ex queue allocated by
          CreateMidi() is freed. The following actions are automatically
          performed by this function:
          A client-supplied sys/ex queue attached with SetSysExQueue() will
          not be freed by this function.
INPUTS    mn - MidiNode to delete. Can be NULL.
RESULTS   None
SEE ALSO  CreateMidi(), ClearSysExQueue()

```

```

camd.library/EndClusterNotify                          camd.library/EndClusterNotify

NAME      EndClusterNotify -- Stop notification of cluster changes
SYNOPSIS  void EndClusterNotify (struct ClusterNotifyNode *cn)
          a0
FUNCTION  Terminates notification of CAMD internal state changes
INPUTS    cn - a pointer to a ClusterNotifyNode
RESULTS   None
SEE ALSO  StartClusterNotify

```

camd.library/FindCluster

camd.library/FindCluster

```
NAME FindCluster -- Find a MidiCluster by name
SYNOPSIS
    struct MidiCluster *FindCluster (name)
                                a0
FUNCTION
    Find a MidiCluster by name. If a node of that name exists, the
    function returns NULL. Midi links (CD_Linkages) must be locked when
    called.
INPUTS
    name - name of MidiCluster to find.
RESULTS
    result - A MidiCluster or NULL.
```

camd.library/FindMidi

camd.library/FindMidi

```
NAME FindMidi -- Find a MidiNode by name
SYNOPSIS
    struct MidiNode *FindMidi (name)
                                a0
FUNCTION
    Find a MidiNode by name. If a node of that name exists, the function
    returns NULL. Midi links (CD_Linkages) must be locked when called.
INPUTS
    name - name of MidiNode to find.
RESULTS
    result - A MidiNode or NULL.
```

cand.library/FlushMidi

cand.library/GetMidi

NAME FlushMidi -- Dispose of all pending messages.

SYNOPSIS
void FlushMidi (struct MidiNode *mi)
a0

FUNCTION
Disposes of all messages waiting to be received by GetMidi() and GetSysEx(). Also clears pending errors.

INPUTS
mi - MidiNode to flush.

RESULTS
None

NAME GetMidi -- Get next MidiMag from buffer.

SYNOPSIS
BOOL GetMidi (struct MidiNode *mn, MidiMag *mag)
a0 a1

FUNCTION
Gets the next MidiMag from mn->MagQueue.
It is definitely not safe to call this if there's no MagQueue.

INPUTS
mn - pointer to MidiNode.
mag - pointer to buffer to place MidiMag removed from queue.

RESULTS
TRUE if a MidiMag was actually copied in mag. FALSE if the buffer was empty.

SEE ALSO
WaitMidi()

International DevCon 93

28

Introduction to CAMD

camd.library/GetMidiAttrA camd.library/GetMidiAttrA

NAME GetMidiAttrA -- Get the attributes of a MidiNode

SYNOPSIS
 ULONG GetMidiAttrA (struct MidiNode *ml, struct TagItem *attr)
 a0 a1

FUNCTION
 Gets attributes of a MidiNode.

INPUTS
 ml - a pointer to the MidiNode
 attr - Attributes to get, terminated with TAG_DONE. The data
 element of each pair contains the address of the storage
 variable.

RESULTS
 Count of attributes understood.

NOTE Not implemented yet. Does nothing at this time.

EXAMPLES

SEE ALSO
 SetMidiAttrA()

camd.library/GetMidiErr camd.library/GetMidiErr

NAME GetMidiErr -- Read accumulated MIDI error flags.

SYNOPSIS
 USHORT GetMidiErr (struct MidiNode *mn)
 a0

FUNCTION
 Returns the current MIDI error flags from the MidiNode. The error
 flags are cleared after reading. Some error flags, such as
 CREF_BufferFull, prevent additional MIDI reception and must be
 cleared in order to restart MIDI reception.

Only bits enabled by SetMidiErrFilter() are returned as 1 bits.

INPUTS
 mn - MidiNode to check.

RESULTS
 CREF_error flags or 0 if no errors were present. The upper 24 bits
 are cleared for the caller's convenience.

SEE ALSO
 SetMidiErrFilter(), WaitMidi()

```

camd.library/GetMidLinkAttrA      camd.library/GetMidLinkAttrA

NAME      GetMidLinkAttrA -- Get attributes of a MidLink
SYNOPSIS  ULONG GetMidLinkAttrA (struct MidLink *ml, struct TagItem *attrs)
          a0          a1
FUNCTION  Gets an attribute of a MidLink.
INPUTS   ml - a pointer to the MidLink
          attrs - Attributes to get, terminated with TAG_DONE. The data
          element of each pair contains the address of the storage
          variable.
RESULTS  Count of attributes understood.
NOTE     Not implemented yet. Does nothing at this time.
EXAMPLES
SEE ALSO  setMidLinkAttrA()

```

```

camd.library/GetSysEx      camd.library/GetSysEx

NAME      GetSysEx -- Read bytes from Sys/Ex buffer.
SYNOPSIS  ULONG GetSysEx (struct MidNode *mn, UBYTE *Buf, ULONG Len)
          a0          a1          d0
FUNCTION  Reads bytes from the Sys/Ex buffer for the current sys/ex message
into the supplied buffer. This function will not read past the end
of the current sys/ex message. The actual number of bytes read is
returned. 0 is returned when there are no more bytes to be read for
the current sys/ex message. If the current message is not a sys/ex
message, this function will return 0.
INPUTS   mn - Pointer to MidNode.
          Buf - Output buffer pointer.
          Len - Max number of bytes to read.
RESULTS  Actual number of bytes read.
SEE ALSO  GetMidl(), SkipSysEx(), QuerySysEx()

```

camd.library/lockcmd camd.library/lockcmd

```

NAME      LockCMD -- Prevent other tasks from changing internal structures
SYNOPSIS  LockCMD(locktype)
          do
          AFTER LockCMD( ULONG );
FUNCTION  This routine will lock the internal semaphores in the CAMD library.
          If they are already locked by another task, this routine will wait
          until they are free.
INPUTS    locktype -- which internal list will be locked.
          CL_LINKS -- locks the internal list of MidiInterfaces, MidiLinks
          and MidiClusters. Any functions that create or delete
          these structures (including SetMidiLinkAttr) will be locked.
RESULT    If locktype is valid, returns a value that must be passed later
          to UnlockCMD.
EXAMPLE
NOTES
BUGS
SEE ALSO  UnlockCMD(), CreateMidi(), DeleteMidi(),
          AddLink(), RemoveMidiLink(), SetMidiLinkAttr();

```

camd.library/midlinkconnected camd.library/midlinkconnected

```

NAME      MidLinkConnected -- Determine if MidLink has a connection
SYNOPSIS  BOOL MidLinkConnected (struct MidLink *ml)
          a0
FUNCTION  Returns TRUE if a MidLink can currently send or receive to anyone.
          For a MidLink of type MLTYPE_Sender, then there must be at least
          one MidLink of type MLTYPE_Receiver linked to the same MidCluster,
          and vice versa.
INPUTS    ml - MidLink to check
RESULTS   result - TRUE if there is a connection, FALSE otherwise

```

```

camd.library/MidiMsgLen                                camd.library/MidiMsgLen

NAME      MidiMsgLen -- Determine the length of a MIDI message
SYNOPSIS  WORD MidiMsgLen (ULONG StatusByte)
          d0
FUNCTION  Returns the length in bytes of the MIDI message described by the
          supplied status byte. The message length includes the status byte.
INPUTS    StatusByte - UBYTE containing status byte.
RESULTS   length - length of the message in bytes. For valid messages this
          will be at least 1. 0 is returned for invalid messages,
          MS_Byte and MS_EOF.
          The result is sign extended to 32 bits for assembly
          programmers.
SEE ALSO  MidiMsgType()

```

```

camd.library/MidiMsgType                                camd.library/MidiMsgType

NAME      MidiMsgType -- Determine the type of a MIDI message
SYNOPSIS  WORD MidiMsgType (MidiMsg *Msg)
          a0
FUNCTION  Returns a type bit number (CMB_) for supplied MidiMsg.
INPUTS    Msg - Pointer to a MidiMsg.
RESULTS   CMB_ bit number of message type. -1 if message is undefined. Also
          -1 is returned for MS_EOF since it's not legal to be passed through
          PutMidi(). The result is sign extended to 32 bits for assembly
          programmers.
SEE ALSO  MidiMsgLen()

```

```

camd.library/NextCluster                                camd.library/NextCluster
NAME
    NextCluster -- Get next MidCluster
SYNOPSIS
    struct MidCluster *NextCluster (struct MidCluster *last)
    a0
FUNCTION
    Returns the next MidCluster on CAMD MidCluster list. If last
    is NULL, returns the first MidCluster. Returns NULL if no more
    MidClusters. Midi links (CD_Linkages) must be locked when called.
INPUTS
    last - previous MidCluster or NULL to get first MidCluster
RESULTS
    next - next MidCluster or NULL

```

```

camd.library/NextClusterLink                            camd.library/NextClusterLink
NAME
    NextClusterLink -- Get next MidLink of one type in a MidCluster
SYNOPSIS
    struct MidLink *NextClusterLink (struct MidCluster *mc,
    a0
    struct MidLink *last, LONG type)
    a1
    d0
FUNCTION
    Returns the next MidLink of a particular type a MidCluster's list
    of MidLinks. If last is NULL, returns the first MidLink. Returns
    NULL if no more MidLinks. Midi links (CD_Linkages) must be locked
    when called.
INPUTS
    last - previous MidLink or NULL to get first MidLink
RESULTS
    next - next MidLink or NULL

```



```

camd.library/NextMidi      camd.library/NextMidi

NAME      NextMidi -- Get next MidiNode

SYNOPSIS
    struct MidiNode *NextMidi (struct MidiNode *last)
                                a0

FUNCTION
    Returns the next MidiNode on CAMD MidiNode list. If last is NULL,
    returns the first MidiNode. Returns NULL if no more MidiNodes.
    Midi links (CD_Linkages) must be locked when called.

INPUTS
    last - previous MidiNode or NULL to get first MidiNode

RESULTS
    next - next MidiNode or NULL

```

```

camd.library/NextMidLink  camd.library/NextMidLink

NAME      NextMidLink -- Get next MidLink of one type in a MidiNode

SYNOPSIS
    struct MidLink *NextMidLink (struct MidiNode *mi,
                                a0
                                struct MidLink *last, LONG type)
                                a1

FUNCTION
    Returns the next MidLink of a particular type in a MidiNode's list
    of MidLinks. If last is NULL, returns the first MidLink. Returns
    NULL if no more MidLinks. Midi links (CD_Linkages) must be locked
    when called.

INPUTS
    last - previous MidLink or NULL to get first MidLink
    type - MLTYPE_Sender or MLTYPE_Receiver

RESULTS
    next - next MidLink or NULL

```

camd.library/OpenMidiDevice camd.library/OpenMidiDevice

NAME OpenMidiDevice -- Open a MIDI device driver.

SYNOPSIS
struct MidiDeviceData *OpenMidiDevice (UBYTE *Name)
a0

FUNCTION
Opens a MIDI device driver.

This function should only be called by Preferences program that wishes to interrogate a particular MIDI device driver for such information as number of ports.

Otherwise this is a private camd.library function.

INPUTS

Name - name of MIDI device driver.

RESULTS

Pointer to MidiDeviceData structure or NULL on failure.

SEE ALSO

CloseMidiDevice ()

camd.library/ParseMidi camd.library/ParseMidi

NAME ParseMidi -- Parse a stream of bytes as MIDI messages.

SYNOPSIS
void ParseMidi (struct MidiLink *ml, UBYTE *Buffer,
a0 a1

ULONG Length)
d0

FUNCTION

Parses an unformatted stream of bytes as MIDI data and transmits the resulting messages to ml->SendPort.

SetMidiLinkAttrs() must be called with MLINK_Parse set to TRUE prior to calling ParseMidi() in order to attach the necessary parser data to the MidiLink.

INPUTS

ml - a MidiLink to send on.

Buffer - Pointer to buffer containing MIDI data to send.

Length - Number of bytes in Buffer to send.

RESULTS

None

SEE ALSO

SetMidiLinkAttrs()

```

camd.library/PutMidi                                camd.library/PutMidi
NAME      PutMidi -- Send a MidiMsg to an output link
SYNOPSIS  void PutMidi (struct MidiLink *link, ULONG Msg)
          a0
FUNCTION   Sends the a MidiMsg to the output link specified.
          The message is automatically timestamped.
INPUTS    Msg - mm_Msg long word component of a MidiMsg.
RESULTS   None
NOTE      Although this function doesn't require a MidiMode pointer, the
          caller must have allocated a MidiMode using CreateMidi() in
          order for any messages to be distributed.
SEE ALSO  PutMidi()

```

```

camd.library/PutMidiMsg                            camd.library/PutMidiMsg
NAME      PutMidiMsg (MACRO) -- Send a MidiMsg.
SYNOPSIS  void PutMidiMsg (struct MidiLink *ml, MidiMsg *msg)
          a0 a1
FUNCTION   Sends a MidiMsg. This is a macro that calls PutMidi().
          The value in msg->mm_port is ignored.
          This macro exists in both C and Assembly.
INPUTS    ml - pointer to MidiLink.
          msg - pointer to a MidiMsg to send. Only the mm_Msg segment is
          used.
RESULTS   None
SEE ALSO  GetMidi(), PutMidi()

```

```

camd.library/PutSysEx                                camd.library/PutSysEx

NAME
    PutSysEx -- Send a Sys/Ex Message to a link.

SYNOPSIS
    void PutSysEx (struct Midilink *ml, UBYTE *Buffer)
        a0

FUNCTION
    Sends the sys/ex message to the specified output link.

    sys/ex messages are sent to hardware units regardless of transmit
    buffer size. Distribution pauses until the entire sys/ex message
    is placed in the transmit queue.

    Distribution to Midinodes is a bit different. If the sys/ex message
    is greater than the Midinode's sys/ex buffer, then the message will
    not be sent to the Midinode. If the message is smaller than the
    Midinode's sys/ex buffer, but there's not enough room left in the
    buffer to contain the sys/ex message, CMF SysExFull will be sent
    to the Midinode. Otherwise the sys/ex message will be sent.

INPUTS
    ml - Midilink to send Sys/Ex to
    Buffer - Pointer to a Sys/Ex message beginning with MS_sysEx and
            ending with MS_EX.

RESULTS
    None

NOTE
    Although this function doesn't require a Midinode pointer, the
    caller must have allocated a Midinode using CreateMidilink() in
    order for any messages to be distributed.

SEE ALSO
    PutSysEx()

```

```

camd.library/QuerySysEx                              camd.library/QuerySysEx

NAME
    QuerySysEx -- Get number of bytes remaining in current Sys/Ex
    message.

SYNOPSIS
    ULONG QuerySysEx (struct Midinode *mn)
        a0

FUNCTION
    Returns the number of bytes remaining in the current sys/ex message.

INPUTS
    mn - pointer to Midinode

RESULTS
    Remaining bytes in sys/ex message. 0 is returned if the last
    message read from GetMidilink() wasn't a sys/ex message.

SEE ALSO
    GetSysEx(), GetMidilink()

```

camd.library/RemoveMidLink camd.library/RemoveMidLink

NAME RemoveMidLink -- Removes a MidLink from a MidCluster.

SYNOPSIS
void RemoveMidLink (struct MidLink *ml)
 a0

FUNCTION
Removes a MidLink structure from a MidCluster.

INPUTS
ml - MidLink to remove. Can be NULL.

SEE ALSO
AddMidLink()

camd.library/RethinkCMD camd.library/RethinkCMD

NAME RethinkCMD -- Force CMD library to reload MIDI preferences

SYNOPSIS
LONG RethinkCMD(void)

FUNCTION
Forces CMD library to reload MIDI preferences and reassign the hardware unit MidLinks.

RESULTS
result - 0 if everything went OK, else returns a CME_error code

```

camd.library/setMidlAttrs      camd.library/setMidlAttrs

NAME      setMidlAttrs -- Set the attributes of a MidiNode

SYNOPSIS
  BOOL setMidlAttrsA (struct MidiNode *mi, struct TagItem *tags)
  a0
  BOOL setMidlAttrs (struct MidiNode *mi, ...)

FUNCTION
  Sets the attributes of a MidiNode, using a Tag List.

  The first form of the function expects a tag array pointer or NULL.
  The second form permits the tag items to exist on the caller's
  stack. In both cases, the final tag item must be TAG_END.

INPUTS
  mi - a pointer to the MidiNode

TagList - optional pointer to tag array. May be NULL. For
OS vi.3, there are restrictions on the tag array
contents. See NOTE below.

TAGS
  MIDI_Name      STRPTR - ti_Data points to the new name of
                    the node (generally the Application name)
  MIDI_SignalTask struct Task * - the task to be signaled whenever
                    a MidiMsg or Participant change occurs. This is set
                    by CreateMidi to the current task as a default
  MIDI_RecvHook  struct Hook * - this hook will be called when
                    new MidiMsgs arrive, if the buffer was empty. If the buffer
                    was not yet empty, then it is simply added onto the end.
  MIDI_PartHook  struct Hook * - this hook will be called whenever
                    any of the clusters that this node is linked to either
                    adds or removes a member.
  MIDI_RecvSignal BYTE * - the signal to send whenever an incoming
                    MidiMsg arrives in the buffer, or -1 to send no signal
  MIDI_PartSignal BYTE * - the signal to send whenever a cluster
                    to which this node is linked has a participant change,
                    or -1 to send no signal
  MIDI_MsgQueue  ULONG * - ti_Data specifies the size of the
                    MsgQueue for this MidiNode. An additional padded MidiMsg
                    is allocated for overflow protection. It can also be
                    set to zero to indicate that no buffer should be allocated
                    (A send-only MidiNode)
  MIDI_sysExSize ULONG * - ti_Data specifies the size of the sysExQueue
                    in bytes. Like the MsgQueue, an additional byte is
                    allocated to allow for overflow protection.
  MIDI_TimeStamp ULONG * - if non-NULL, ti_Data is a pointer to
                    a longword which is to be used as the source for time
                    stamps of incoming MidiMsgs for this MidiNode. It is
                    assumed that the longword pointed to will be updated by
                    some other mechanism -- for example, the longword could
                    point to one of the fields in a PlayerInfo structure.
  MIDI_ErrFilter UWORD * - specified the ErrFilter for this MidiNode
  MIDI_ClientType UWORD * - specified the Client Type for this MidiNode
                    See camd.h for more detail.
  MIDI_Image      struct Image * - pointer to an Intuition Image

```

```

structure representing a glyph or icon that is symbolic
if this application. Will be used for a future "patch
bay" application. It is suggested that images be
approximately 32 wide x 32 high, for consistency.

RESULTS
  TRUE if all changes were made successfully or FALSE on failure.
  When FALSE is returned, an error code will be returned if a TagItem
  with an ti_Tag of MIDI_ErrorCode was provided. The CMS_error code
  will be put in the ti_Data field.

NOTE
  Under 1.3 only a restricted tag array may be passed into this
  function. Specifically, the only special tag values that are
  supported are TAG_END (or TAG_DONE) and TAG_IGNORE. All others
  (e.g. TAG_SKIP, TAG_MORE) are treated as TAG_END. The full range of
  TAG list operations are permitted under 2.0.

  This function MUST be called by a Process (not a Task). It makes a
  few DOS calls.

  Don't call WaitMidi() with a MidiNode that has no ReceiveSignal.

EXAMPLES
  mi = SetMidlAttrs (mi,
                    MIDI_MsgQueue, 2048, MIDI_sysExSize, 10000L, TAG_END);

  Modifies a MidiNode to have space to receive 2048 MidiMsg's and 10000
  bytes of SysEx data. Note the 'L' on the end of 10000. This
  forces correct alignment of the tag items on the stack for 16 bit
  integer compilation.

  SEE ALSO
  CreateMidi(), DeleteMidi()
  2.0 tag docs.

```

camd.library/SetMidLinkAttrs camd.library/SetMidLinkAttrs

NAME SetMidLinkAttrs -- Set the attributes of a MidLink

SYNOPSIS
 BOOL SetMidLinkAttrs (struct MidLink *ml, struct TagItem *tags)
 a0 a1

BOOL SetMidLinkAttrs (struct MidLink *ml, Tag type, ...)

FUNCTION
 Sets the attributes of a MidLink, using a Tag List.

The first form of the function expects a tag array pointer or NULL. The second form permits the tag items to exist on the caller's stack. In both cases, the final tag item must be TAG_END.

INPUTS
 ml - a pointer to the MidLink

TagList - optional pointer to tag array. May be NULL. For OS v1.3, there are restrictions on the tag array contents. See NOTE below.

TAGS
 MLINK_Name STRPTR - ti_Data points to the new name of the node
 MLINK_Location STRPTR - ti_Data points to name of MidLinkCluster to link with
 MLINK_ChannelMask UWORD - ti_Data contains mask of which MIDI Channels to listen for (defaults to -0)
 MLINK_EventMask UWORD - ti_Data contains mask of which types of MIDI events to listen for (defaults to -0)
 MLINK_UserData CPTR - ti_Data points to user definable data

STRPTR - ti_Data points to a comment string. The highest priority MidLink in a MidLinkCluster has its comment field copied to the MidLinkCluster's comment field

MLINK_PortID UBYTE - ti_Data contains value to copy into any MidLink's arriving at MidLink through this MidLink (defaults to 0)

MLINK_Private BOOL - if ti_Data contains TRUE, then this link requests to not be shown by "patch bay" editors, etc.

MLINK_Priority BYTE - ti_Data contains priority of the MidLink
 MLINK_SysExFilter ULONG - ti_Data contains three 1-byte manufacturer numbers to filter SysEx messages with

MLINK_SysExFilterX ULONG - ti_Data contains one 3-byte manufacturer number to filter SysEx messages with

MLINK_Parse BOOL - if ti_Data contains true, allocate a parser for the MidLink so raw MIDI streams can be sent through the link

MLINK_ErrorCode ULONG * - ti_Data points to an error code buffer

RESULTS
 TRUE if all changes were made successfully or FALSE on failure. When FALSE is returned, an error code will be returned if a TagItem with an ti_Tag of MLINK_ErrorCode was provided. The CME_error code will be put in the ti_Data field.

NOTE

Under 1.3 only a restricted tag array may be passed into this function. Specifically, the only special tag values that are supported are TAG_END (or TAG_DONE) and TAG_IGNORE. All others (e.g. TAG_SKIP, TAG_MORE) are treated as TAG_END. The full range of TAG list operations are permitted under 2.0.

EXAMPLES

```
mi = SetMidLinkAttrs (ml,  
                      MLINK_Location, "out.0", MLINK_Priority, -5L, TAG_END);
```

Modifies a MidLink so it will be connected to the MidLinkCluster named "out.0" and makes the MidLink's priority -5. Note the 'L' on the end of -5. This forces correct alignment of the tag items on the stack for 16 bit integer compilation.

SEE ALSO

AddMidLink(), RemoveMidLink()
 2.0 tag docs.

```

camd.library/skipSysEx                                camd.library/skipSysEx

NAME
    skipSysEx -- skip the next sys/ex message in the Sys/Ex buffer.

SYNOPSIS
    void skipSysEx (struct MidlMode *mn)
        a0

FUNCTION
    Skips the remaining bytes of the current sys/ex message. Nothing
    happens if the last message read from Gethid() wasn't a sys/ex
    message.

INPUTS
    mn - Pointer to MidlMode.

RESULTS
    None.

SEE ALSO
    GethSysEx()

```

```

camd.library/startClusterNotify                        camd.library/startClusterNotify

NAME
    StartClusterNotify -- Notify task of cluster changes

SYNOPSIS
    void startClusterNotify (struct ClusterNotifyMode *cn)
        a0

FUNCTION
    Allow an external task to receive notification via a signal that
    CAMD's internal state has changed. This function is mostly useful
    to "patch bay" editors, etc.

INPUTS
    cn - a pointer to a ClusterNotifyMode

RESULTS
    None

EXAMPLES
    struct ClusterNotifyMode cnn;
    cnn.cnn_Task = FindTask(NULL);
    cnn.cnn_sigBit = AllocSignal(-1);
    StartClusterNotify(&cnn);
    /* later in code */
    Wait(1 << cnn.cnn_sigBit);
    /* someone changed the state of a cluster */

SEE ALSO
    EndClusterNotify

```


camd.library/UnlockCAMD

camd.library/WaitMidi

NAME UnlockCAMD -- Unlock internal lists

SYNOPSIS
UnlockCAMD(lock)
a0
void UnlockCAMD(APTR);

FUNCTION
Undoes the effects of LockCAMD().

INPUTS
lock -- value returned by LockCAMD(). Can be NULL.

EXAMPLE

SEE ALSO
LockCAMD()

NAME WaitMidi -- Wait for next MidiMsg and get it.

SYNOPSIS
BOOL WaitMidi (struct MidiNode *mn, MidiMsg *msg)
a0 a1

FUNCTION
Waits for the next MidiMsg to arrive at the MidiNode. On reception, the MidiMsg is removed and copied to the supplied buffer.

WaitMidi() first checks for errors detected at the MidiNode. If none are detected, GetMidi() is called. If the buffer was empty, Wait() is called until the MidiNode is signalled. Once the signal arrives, this process is repeated.

INPUTS
mn - pointer to MidiNode.
msg - pointer to buffer to place MidiMsg removed from queue.

RESULTS
TRUE if a MidiMsg was received. FALSE if an error was detected.

SEE ALSO
GetMidi() OO

Selected Realtime Library Header Files

```

realtime.h
$ifndef MIDI_REALTIME_H
$define MIDI_REALTIME_H
/*.....*/
* Realtime Library (timing & syncing system)
*.....*/
* Design & Development - Talin & Joe Pearce
*
* Copyright 1992 by Commodore Business Machines
*.....*/
* realtime.h - Realtime conductor/player include file.
*.....*/
$ifndef EXEC_LISTS_H
$include <exec/lists.h>
$endif
$ifndef EXEC_TYPES_H
$include <exec/types.h>
$endif
$ifndef UTILITY_TAGITEM_H
$include <utility/tagitem.h>
$endif
$ifndef UTILITY_HOOKS_H
$include <utility/hooks.h>
$endif
/* Each Conductor represents a group of applications which wish to remain
* synchronized together.
*/
struct Conductor {
    struct Node cdt_Node; /* conductor list node */
    WORD cdt_NodeId;
    struct MinList cdt_Players; /* list of players */
    ULONG cdt_ClockTime; /* current time of this sequence */
    cdt_StartTime; /* start time of this sequence */
    cdt_EndTime; /* time from external unit */
    cdt_MaxExternalTime; /* upper limit on sync'd time */
    cdt_Metronome; /* Metronome */
    cdt_PlayersNotReady; /* count of players not ready */
    cdt_Flags; /* clock flags */
    cdt_State; /* playing or stopped */
    cdt_Stopped; /* quick stopped check */
};

$define CONDUCTF_INTERVAL(1<<0) /* clock is externally driven */
$define CONDUCTF_COTICK(1<<1) /* received lat external tick */
$define CONDUCTF_METROSET(1<<2) /* Metronome filled in */

enum time_states {
    CLOKSTATE_STOPPED=0, /* clock is stopped */
    CLOKSTATE_PAUSED, /* clock is paused */
    CLOKSTATE_LOCATE, /* go to 'running' when ready */
    CLOKSTATE_RUNNING, /* run clock NOW */
    CLOKSTATE_METRIC=1, /* ask high node to locate */
    CLOKSTATE_SHUTTLE=2, /* time changing but not running */
};

$define CONDSTOPF_STOPPED(1<<0) /* not running state */
$define CONDSTOPF_NOTICK(1<<1) /* extsync & not gottick */

```

```

/* The PlayerInfo is the connection between a Conductor and an application.
*/
struct PlayerInfo {
    struct Node pi_Node; /* callback for player */
    WORD pi_NodeId;
    struct Hook pi_Hook; /* callback for player */
    struct Conductor pi_Source; /* pointer to parent context */
    struct Task pi_Task; /* task to signal for alarm */
    LONG pi_MetricTime; /* current time in app's metric */
    LONG pi_AlarmTime; /* time to wake up */
    void pi_UserData; /* for application use */
    WORD pi_PlayerID; /* for application use */
    WORD pi_Flags; /* general PlayerInfo flags */
    ULONG pi_Reserved[2]; /* internal use */
    BYTE pi_AlarmSigBit; /* signal to send for alarms */
};

$define PLAYERF_READY(1<<0) /* player is ready to go */
$define PLAYERF_ALARMSET(1<<1) /* alarm is set */
$define PLAYERF_QUIET(1<<2) /* a dummy player, used for sync */
$define PLAYERF_CONDUCTED(1<<3) /* give me metered time */
$define PLAYERF_ENTSYNC(1<<4) /* granted external sync */

$define PLAYER_Base (TAG USER+64)
$define PLAYER_Base_1 (PLAYER_Base+1) /* set address of hook function */
$define PLAYER_Base_2 (PLAYER_Base+2) /* name of player */
$define PLAYER_Base_3 (PLAYER_Base+3) /* priority of player */
$define PLAYER_Base_4 (PLAYER_Base+4) /* set conductor for player */
/* if -0, create private conductor */
$define PLAYER_Base_5 (PLAYER_Base+5) /* the "ready" flag */
$define PLAYER_Base_6 (PLAYER_Base+6) /* task to signal for alarm/notify */
$define PLAYER_Base_7 (PLAYER_Base+7) /* sets/clears CONDUCTED flag */
$define PLAYER_Base_8 (PLAYER_Base+8) /* signal bit for alarm (or -1) */
$define PLAYER_Base_9 (PLAYER_Base+9) /* don't process time thru this */
$define PLAYER_Base_10 (PLAYER_Base+10)
$define PLAYER_Base_11 (PLAYER_Base+11)
$define PLAYER_Base_12 (PLAYER_Base+12) /* alarm time (sets PLAYERF_ALARMSET) */
$define PLAYER_Base_13 (PLAYER_Base+13) /* sets/clears ALARMSET flag */
$define PLAYER_Base_14 (PLAYER_Base+14) /* attempt/release to ext sync */
$define PLAYER_Base_15 (PLAYER_Base+15) /* error return value */

/* Messages sent via PlayerInfo hook.
*/
enum player_methods {
    PM_TICK=0,
    PM_STATE,
    PM_POSITION,
    PM_SHUTTLE
};

struct pmtime {
    ULONG pmt_Method; /* used for PM_TICK, PM_POSITION and PM_SHUTTLE */
    ULONG pmt_Time;
};

struct pmsstate {
    ULONG pms_Method; /* used for PM_STATE */
    ULONG pms_OldState;
};

enum {
    RT_Conductors, /* conductor list */
    RT_Locks
};

```

```

/* *****
* RealTime Error Codes
* *****
*
* Define RTE_MemMan 801 /* memory allocation failed */
* Define RTE_Mosignals 802 /* signal allocation failed */
* Define RTE_Motimer 803 /* timer (CIA) allocation failed */
* Define RTE_Playing 804 /* Can't shuttle while playing */
*
* Endif /* MIDI_REALTIME_H */
*/

```

```

realtimebase.h
#define MIDI_REALTIMEBASE_H
#define MIDI_REALTIMEBASE_H
/* *****
* RealTime Library (timing & syncing system)
* *****
* Design & Development - Talin & Joe Pearce
* Copyright 1992 by Commodore Business Machines
* *****
* realtimebase.h - RealTime library base structure
* *****
*
* #ifndef EXEC_LIBRARIES_H
* #include <exec/libraries.h>
* #endif
*
* #ifndef EXEC_LISTS_H
* #include <exec/lists.h>
* #endif
*
* #ifndef EXEC_SEMAPHORES_H
* #include <exec/semaphores.h>
* #endif
*
* struct RealTimeBase
* {
*     struct Library Lib;
*     ULONG pad0;
*
*     /* publicly available stuff (READ ONLY) */
*     ULONG Time; /* current time */
*     ULONG TimeDelta; /* time delta */
*     WORD TickFreq; /* ideal RealTime Tick frequency */
*     WORD TickErr; /* nanosecond error from ideal Tick length to real tick length
*
*     /* actual tick length is: 1/TickFreq + TickErr/1e9 */
*     /* -705 < TickErr < 705 */
*
*     /* private stuff below here */
* };
*
* #define RealTime_TickErr_Min -705
* #define RealTime_TickErr_Max 705
* #endif

```

Introduction to CAMD

* The default Unit Ports are:

```
struct MidiCluster {
    struct Node
        mcl_Node; /* linked list node */
        mcl_Participants; /* list of receivers */
        struct List
            mcl_Receivers, /* list of receivers */
            mcl_Senders, /* list of senders */
            mcl_PublicParticipants; /* if >, cluster is public */
        WORD
            mcl_Flags; /* flag for this location */
        WORD
            /* NOTE: Cluster name follows structure, and is pointed to by ln_Name */
};
```

```

/*-----
* MidLink -- links a cluster and a MidiNode
*
* All fields are READ ONLY. Modifications to fields may
* be performed only through the appropriate library function
* calls.
*-----
*/
struct MidLink {
    struct Mode;
    struct MinMode;
    struct MidiNode;
    struct MidiCluster;
    char *ml_ClusterComment;
    BYTE ml_Flags;
    BYTE ml_PortID;
    WORD ml_ChannelMask;
    ULONG ml_EventTypeMask;
    union SysExFilter {
        BYTE match_id(s);
        ULONG xsf_Packed;
        struct SysExFilter;
    } ml_SysExFilter;
    APTR ml_ParserData;
    APTR ml_UserData;
};

/* SysExFilter members */
#define xsf_Mode b[0] /* SXM mode bits */
#define xsf_ID1 b[1] /* 3 1-byte id's or 1 3-byte id */
#define xsf_ID2 b[2]
#define xsf_ID3 b[3]

/* MidLink types */
enum {
    MLTYPE_Receiver,
    MLTYPE_Sender,
    MLTYPE_Bytes
};

/* ml_Flags */
#define MLF_Sender (1<<0)
#define MLF_PartChange (1<<1)
#define MLF_PrivateLink (1<<2)
#define MLF_DeviceLink (1<<3)

/* MidLink tags */
#define TAG_USER+65

/* MLINK locations */
#define MLINK_Location (MLINK_Base+0)
#define MLINK_ChannelMask (MLINK_Base+1)
#define MLINK_EventMask (MLINK_Base+2)
#define MLINK_UserData (MLINK_Base+3)
#define MLINK_PortID (MLINK_Base+4)
#define MLINK_Private (MLINK_Base+5)
#define MLINK_Priority (MLINK_Base+6)
#define MLINK_Filter (MLINK_Base+7)
#define MLINK_FilterChange (MLINK_Base+8)
#define MLINK_SysExFilter (MLINK_Base+9)
#define MLINK_Parse (MLINK_Base+10)
#define MLINK_Reserved (MLINK_Base+11)
#define MLINK_ErrorCode (MLINK_Base+12)

/*-----
* Bit packed as follows: 00000mcc
* m - mode bit
* cc - count bits 0 - 3 (only used for SXM_1byte)
*-----
*/
#define SXM_ModeBits 0x04 /* mask for count bits for SXM_1byte */
#define SXM_CountBits 0x03
#define SXM_Off 0x00 /* don't filter (equal to SXM_1byte | 0) */
#define SXM_1byte 0x00 /* match upto 3 1-byte id's */
#define SXM_3byte 0x04 /* match a single 3-byte id */

/*-----
* MidiNode
*
* All fields are READ ONLY. Modifications to fields may
* be performed only through the appropriate library function
* calls.
*-----
*/
struct MidiNode {
    struct Mode;
    struct Image;
    struct MinList;
    struct Task;
    struct Hook;
    BYTE ml_SigTask;
    BYTE ml_ReceiveSigBit;
    BYTE ml_ParticipantSigBit;
    BYTE ml_ErrFilter;
    BYTE ml_Alignment[1];
    ULONG ml_TimeStamp;
    ULONG ml_MsgQueueSize;
    ULONG ml_SysExQueueSize;
};

/* linked list node */
/* type of application */
/* image for patch panel */
/* list of output links */
/* list of input links */
/* task to signal */
/* hook (and list node) */
/* hook for participant change */
/* signalmask for new data */
/* signalmask for part change */
/* CMEF_error filter for */
/* for longword alignment */
/* where timestamps come from */
/* size of buffers */

/* private stuff below here */
/* client types */
#define CCType_Sequence (1<<0)
#define CCType_SamplerEditor (1<<1)
#define CCType_PatchEditor (1<<2)
#define CCType_Motator (1<<3)
#define CCType_EventProcessor (1<<4)
#define CCType_EventFilter (1<<5)
#define CCType_EventRouter (1<<6)
#define CCType_ToneGenerator (1<<7)
#define CCType_EventGenerator (1<<8)
#define CCType_GraphicsAnimator (1<<9)
/* e.g MIDI thru task */
/* i.e. using Amiga to make */
/* e.g algorithmic composition */
/* e.g syncing animation */
/* Tags for CreateMidi() and SetMidiAttrs() */
#define MIDI_Base (TAG_USER+65)

```

```

#define MIDI_Name (MIDI_Base+0) /* name of application */
#define MIDI_SignalTask (MIDI_Base+1) /* task to signal if not this one */
#define MIDI_RecvHook (MIDI_Base+2) /* hook for incoming data */
#define MIDI_PartHook (MIDI_Base+3) /* hook for participant change */
#define MIDI_RecvSignal (MIDI_Base+4) /* signal to use if incoming */
#define MIDI_PartSignal (MIDI_Base+5) /* signal to use if incoming */
#define MIDI_MqQueue (MIDI_Base+6) /* size of event buffer */
#define MIDI_SysExSize (MIDI_Base+7) /* size of sysex buffer */
#define MIDI_Timestamp (MIDI_Base+8) /* timer to timestamp with */
#define MIDI_ErrFilter (MIDI_Base+9) /* error filter bits */
#define MIDI_ClientType (MIDI_Base+10) /* client type mask */
#define MIDI_Image (MIDI_Base+11) /* application image */
#define MIDI_ErrorCode (MIDI_Base+12) /* (ULONG *) - Error Code buffer for returning CME_error codes. */

/*****
 * CreateMidi() Error Codes
 *
 * These are the IoErr() codes that CreateMidi() can return
 * on failure.
 *
 * !!! need specific error code set for each function instead!
 *****/

#define CME_NoMem 801 /* memory allocation failed */
#define CME_NoSignals 802 /* signal allocation failed */
#define CME_NoTimer 803 /* timer (CIA) allocation failed */
#define CME_NoDrafs 804 /* badly formed midi.prafs file */

#define CME_NoUnit(unit) (820 + (unit)) /* unit open failure */

#if 0 /* !!! old */
/*****
 * MidMode tag items for use with CreateMidi().
 *****/

#define CMA_base (TAG_USER + 64)
#define CMA_SysEx (CMA_base + 0) /* int - allocate a sys/ex buffer,
 * ti Data specifies size in bytes.
 * Default is 0. Only valid if
 * RecvSize is non-zero. */
#define CMA_Parse (CMA_base + 1) /* bool - enable usage of ParseMidi().
 * Default is FALSE. */
#define CMA_Alarm (CMA_base + 2) /* bool - enable usage of
 * SetMidiAlarm().
 * Default is FALSE. */
#define CMA_SendPort (CMA_base + 3) /* int - Initial SendPort. Default is
 * CME_Out(0). */
#define CMA_PortFilter (CMA_base + 4) /* int - Initial PortFilter. Default is
 * 0. */
#define CMA_TypeFilter (CMA_base + 5) /* int - Initial TypeFilter. Default is
 * 0. */
#define CMA_ChannFilter (CMA_base + 6) /* int - Initial ChannFilter. Default is
 * 0. */

```

```

#define CMA_SysExFilter (CMA_base + 7) /* packed - Initial SysExFilter as
 * returned
 * by one of the PackSysExFilterM()
 * macros.
 * Default is no filtering (i.e. recv
 * all). */
#define CMA_ErrFilter (CMA_base + 8) /* int - Initial ErrFilter. Default is
 * 0. */
#endif

/*****
 * MIDI Message Type Bits
 *
 * Returned by MidiMagType() and used with SetMidiFilters().
 *****/

#define CMB_Note 0
#define CMB_Prog 1
#define CMB_PitchBend 2
#define CMB_CtrlMSB 3
#define CMB_CtrlLSB 4
#define CMB_CtrlSwitch 5
#define CMB_CtrlByte 6
#define CMB_CtrlParam 7
#define CMB_CtrlUnder 8 /* for future ctrl # expansion */
#define CMB_Mode 9
#define CMB_ChannPress 10
#define CMB_PolyPress 11
#define CMB_RealTime 12
#define CMB_SysCom 13
#define CMB_SysEx 14

/* (these need to be long for SetMidiFilters()) */
#define CMF_Note (1L << CMB_Note)
#define CMF_Prog (1L << CMB_Prog)
#define CMF_PitchBend (1L << CMB_PitchBend)
#define CMF_CtrlMSB (1L << CMB_CtrlMSB)
#define CMF_CtrlLSB (1L << CMB_CtrlLSB)
#define CMF_CtrlSwitch (1L << CMB_CtrlSwitch)
#define CMF_CtrlByte (1L << CMB_CtrlByte)
#define CMF_CtrlParam (1L << CMB_CtrlParam)
#define CMF_CtrlUnder (1L << CMB_CtrlUnder)
#define CMF_Mode (1L << CMB_Mode)
#define CMF_ChannPress (1L << CMB_ChannPress)
#define CMF_PolyPress (1L << CMB_PolyPress)
#define CMF_RealTime (1L << CMB_RealTime)
#define CMF_SysCom (1L << CMB_SysCom)
#define CMF_SysEx (1L << CMB_SysEx)

/* some handy type macros */
#define CMF_Ctrl (CMF_CtrlMSB | CMF_CtrlLSB | CMF_CtrlSwitch |
 * CMF_CtrlByte | CMF_CtrlParam | CMF_CtrlUnder)
#define CMF_Channel (CMF_Note | CMF_Prog | CMF_PitchBend | CMF_Ctrl |
 * CMF_Mode | CMF_ChannPress | CMF_PolyPress)
#define CMF_All (CMF_Channel | CMF_RealTime | CMF_SysCom | CMF_SysEx)

```

```

/* MIDI Error Flags
 *
 * These are error flags that can arrive at a MidiMode.
 * An application may choose to ignore or process any
 * combination of error flags. See SetMidErrFilter() and
 * GetMidErr() for more information.
 */
#define CMDB_MidErr 0 /* Invalid message was sent */
#define CMDB_BufferFull 1 /* MidBuffer is full */
#define CMDB_SysExFull 2 /* SysExBuffer is full */
#define CMDB_ParamMem 3 /* sys/ex memory allocation failure during parse */
#define CMDB_RecvErr 4 /* serial receive error */
#define CMDB_RecvOverflow 5 /* serial receive buffer overflow */
#define CMDB_SysExTooBig 6 /* Attempt to send a sys/ex message bigger than
SysExBuffer */

#define CMDB_MidErr (1L << CMDB_MidErr)
#define CMDB_BufferFull (1L << CMDB_BufferFull)
#define CMDB_SysExFull (1L << CMDB_SysExFull)
#define CMDB_ParamMem (1L << CMDB_ParamMem)
#define CMDB_RecvErr (1L << CMDB_RecvErr)
#define CMDB_RecvOverflow (1L << CMDB_RecvOverflow)
#define CMDB_SysExTooBig (1L << CMDB_SysExTooBig)

/* a handy macro for SetMidErrFilter() */
#define CMDB_Err All (CMDB_MidErr | CMDB_BufferFull | CMDB_SysExFull |
CMDB_SysExTooBig | CMDB_ParamMem | CMDB_RecvErr | CMDB_RecvOverflow)

if 0 /* !!! old */
/*
 * MidTickHookMsg
 *
 * Message structure passed to Tick Hooks
 */
struct MidTickHookMsg
{
    ULONG ID; /* msg ID (always MTHM_Tick for now) */
    ULONG Time; /* Time at tick */
};

enum
{
    MTHM_Tick
};

/* MidByteHookMsg
 *
 * Message structure passed to Byte Hooks
 */
struct MidByteHookMsg
{
    ULONG ID; /* msg ID (always MBHM_Byte for now) */
    BYTE UnitNum; /* Midi Unit number where byte was received */
    BYTE pad0; /* Data as it was received by the serial hardware.
    Bits 0-7 are the MIDI byte. Bit 15, when set,
    indicates a hardware receive error. */
    DWORD RecvData;
};

```

```

enum
{
    MBHM_Byte
};

endif
/*
 * Hook Message ID's
 *
 * Each Hook passes as the "msg" param a pointer to one of these (LONG)
 * Can be extended for some types of messages
 */
enum
{
    CMMSG_Recv, /* receive MIDI message */
    CMMSG_Link, /* a linkage notification */
};

/*
 * CMMSG_Link structure
 */
struct cmLink
{
    ULONG CmlMethodID;
    ULONG CmlAction;
};

enum
{
    CACT_Link,
    CACT_Unlink
};

/*
 * ClusterNotifyNode
 */
struct ClusterNotifyNode
{
    struct MinMode cnn_Mode; /* the usual node */
    struct Task cnn_Task; /* task to signal */
    BYTE cnn_sigbit; /* sigbit to use */
    BYTE pad[3];
};

/*
 * CAMD Macros
 *
 * See camd.doc for info.
 */
#define PackSysExFilter0() ((ULONG) SXEM Off << 24)
#define PackSysExFilter1(id1) ((ULONG) (SXEM_lbyte | 1) << 24 |
(ULONG) id1 << 16)
#define PackSysExFilter2(id1, id2) ((ULONG) (SXEM_lbyte | 2) << 24 |
(ULONG) id1 << 16 | (id2) << 8)
#define PackSysExFilter3(id1, id2, id3) ((ULONG) (SXEM_lbyte | 3) << 24 |
(ULONG) id1 << 16 | (id2) << 8 | (id3))
#define PutMidMsg(ml, msg) PutMidMsg((ml), (msg) -> 1(0))

```




DSP On The Amiga

Using The AT&T DSP3210™
and VCOS™ On The Amiga

Document Revision 0.6

1993 DevCon Release

by
Eric Lavitsky
December, 1992

Copyright © 1992 Lavitsky Computer Laboratories, Inc.
All rights reserved worldwide.

This document contains preliminary information and is subject to change without notice.

Disclaimer: While the author(s) of this document have made every attempt to verify its accuracy, the information provided herein is provided "as is" without warranty of any kind, either express or implied. The author(s) and Lavitsky Computer Laboratories, Inc. assume no liability for any damages, direct or indirect, resulting from any defect in this information.

This document assumes an understanding of the Amiga architecture and programming practices.

Amiga is a registered trademark of Commodore-Amiga, Inc. VCOS and DSP3210 are trademarks of AT&T. MS-DOS is a registered trademark of Microsoft.

This document was created entirely on an Amiga 3000 using PageStream 2.2 from Soft-Logik.

I. Introduction

As the demand for high quality digital data in multimedia applications increases, so does the demand to rapidly process the accompanying complex data streams. Existing personal computer CISC and RISC architectures lack the signal processing throughput required to process these data streams efficiently and in real time. Digital Signal Processors (DSPs) are designed to process real-world signals (e.g., audio and video signals) in real time. Recently, the cost of DSPs has dropped to the point that they can be integrated into personal computer systems very effectively. Even in more advanced RISC and workstation systems, DSPs can be used to perform signal processing tasks without degrading the performance of the host CPU.

The Amiga has always been a strong multimedia computer, in fact, many would consider it the first viable multimedia personal computer. Its hardware and software architecture is well suited to processing multiple simultaneous data streams. Amiga designers and system engineers have always had the foresight to know that the existing system would need to grow in order to meet the future needs of its users. This foresight has made it relatively easy to integrate a DSP into the Amiga system.

Many system manufacturers have attempted to integrate DSPs into their platforms in the past, but have met with only limited success. The key factors contributing to this have been the limitations of the DSPs they have used and the lack of flexible and advanced applications that take advantage of them. The AT&T DSP3210 is a very flexible and low-cost DSP capable of computing up to 33 million floating point operations per second (MFLOPS). It can be designed into a host computer with relatively little overhead. AT&T has solved the lack of applications by providing a multitasking operating system called VIOS, as well as a full software library implementing a wide variety of signal processing and multimedia algorithms for the DSP3210. Using the AT&T DSP3210 and VIOS, it is now possible to provide applications such as speech recognition, CD-quality audio (compressed), high-speed telecommunications, high-quality music synthesis and many others with little or no impact on system performance. Even a powerful CPU like the Motorola 68040 could not hope to keep up with the processing demands of even a single one of these applications, let alone several of them simultaneously.

II. Hardware

II.1 Overview

The DSP3210 is a full 32-bit floating point DSP implemented in .9 micron CMOS. It provides many advantages over fixed point DSPs such as the Motorola 56000. Some of the main features of the DSP3210 include:

- 32-bit floating point arithmetic.
- 32-bit addressing.
- Large (8K) on-chip, zero wait-state memory.
- Single cycle instructions (for up to 33Mflops).
- Share bus with Motorola or Intel style CPU.
- Serial I/O with DMA transfer counters for up to 25Mbits/second transfer:
 - Serial data transfers occur without processor intervention.
 - Cycles are stolen when necessary.
 - DMA control for serial in and serial out.
- Barrel Shifter for bit manipulation in graphics or data encryption.
- Both mu-law and A-law encoding.
- Bit I/O general purpose 8-bit I/O port for control of external hardware

- Programmable 32-bit timer for interval timing, rate generation, event counting or waveform generation.
- Fully vectored interrupt structure with hardware context save:
Allows very fast interrupt processing, up to 2 million/second.
- Low power CMOS design.

No special programming is required on the DSP3210 to implement floating point algorithms, or to process signals with a much larger dynamic range (in excess of 1500db as opposed to < 300db for fixed point). The DSP3210 is also designed to share a host memory bus with either a Motorola or Intel style CPU. This greatly reduces system cost by removing the requirement for expensive fast local memory for the DSP. This also removes any practical restrictions on program or data size. A large on-chip cache (8K) combined with software that intelligently utilizes the cache allows the DSP3210 to execute complex signal processing algorithms without expensive local memory. All instructions execute in a single cycle (four clock periods: 80ns for a 50Mhz part or 60ns for a 66Mhz part) and includes all floating point post normalization (which is performed automatically). A single instruction may have two floating point operations: a floating point multiplication and a floating point addition. The DSP3210 also supports up to four memory accesses in a single instruction cycle (quad-word transfer). The DSP3210 architecture features seven functional units:

- Control Arithmetic Unit (CAU)
- Data Arithmetic Unit (DAU)
- On-chip memory (RAM0, RAM1, Boot ROM)
- Bus Interface
- Serial I/O (SIO)
- DMA Controller (DMAC)
- Timer/Status Control (TSC)

II.II The Control Arithmetic Unit

The CAU is responsible for address calculation, branching control and all 16 and 32-bit integer logic and arithmetic operations. It is a RISC core consisting of a 32-bit Arithmetic Logic Unit (ALU), a 32-bit Program Counter (PC), 22 32-bit general purpose registers (r0-r22) and a 32-bit barrel shifter. This core executes instructions at up to 16.7 million instructions per second. There are special register considerations in the CAU:

r0	hardwired to 0 (always)
r1-r14	DA instruction memory reference (X,Y,Z) pointer registers
r15-r19	DA instruction memory reference (X,Y,Z) increment registers
r20	used by error exception facility to store old pc
r21	stack pointer (sp)
r22	pointer to the exception vector table (evtp)

The CAU provides the following branching and control instructions:

if (COND) goto {N, rB, rB+N}	Conditional branch based on flags
if (rM-->=0) goto {N, rB, rB+N}	Conditional branch using loop counter
goto {N, rB, rB+N, M, rB+M}	Unconditional branch
nop	No operation
call {N, rB, rB+N, M}(rM)	Call subroutine
return {rM}	Return from subroutine
do K,{L, rM}	Do next K+1 instruction(s) L+1 or (rM+1)

dolock K,{L, rM	times. K=0,1,2...127; L=rM=0,1,2...2047
doblock {L, rM}	Signals interlocked bus access
ireturn	Signals quad-word transfers
sfrst	Return from interrupt
	Soft-reset; changes error level to base level; encoded as spc=(byte)r0
waiti	Wait for interrupt; encoded as spc=(long)r0

where: rB = pc, r0-r22
rM = r1-r22
N = 16-bit unsigned integer
M = 24-bit unsigned integer
COND = one of the DSP3210 condition codes (refer to DSP3210 manual)

II.III The Data Arithmetic Unit

The DAU consists of a 32-bit floating point multiplier, a 40-bit floating point adder, four 40-bit floating point accumulators (a0-a3), a clip test register (ctr), and a control register (dauc). The multiplier and adder operate in parallel to perform up to 16.7 million computations per second (12.5 million for a 50Mhz part) of the form $(a=b+c*d)$, also known as a multiply-accumulate. The DAU contains a four stage pipeline which is visible to the application programmer. The DAU supports the following floating point formats:

Single precision (32-bit) in both DSP32 and IEEE format
Extended single precision (40-bit) (uses 8 mantissa guard bits)

Single instruction data type conversions are done in the DAU hardware:

DSP32 and IEEE 32-bit floating point
16/32-bit integer
8-bit unsigned
mu-law and A-law

The DAU has a number of special instructions to greatly simplify data type conversions and other common operations:

[Z=] aN = ic(Y)	Input conversion mu-law, A-law, 8-bit linear to float.
[Z=] aN = oc(Y)	Output conversion float to mu-law, A-law, 8-bit linear.
[Z=] aN = float16(Y)	16-bit integer to float.
[Z=] aN = float32(Y)	32-bit integer to float.
[Z=] aN = int16(Y)	Float to 16-bit integer (round or truncate, dauc[4]).
[Z=] aN = int32(Y)	Float to 32-bit integer (round or truncate, dauc[4]).
[Z=] aN = round(Y)	Round to nearest, float(40) to float(32).
[Z=] aN = ifalt(Y)	Conditional assignment/memory write.
[Z=] aN = ifaeq(Y)	Conditional assignment/memory write.
[Z=] aN = ifagt(Y)	Conditional assignment/memory write.
[Z=] aN = dsp(Y)	IEEE to DSP format conversion.
[Z=] aN = ieee(Y)	DSP to IEEE format conversion.
[Z=] aN = seed(Y)	32-bit to 32-bit reciprocal seed.

Where [Z=] indicates that condition codes may be set. Note that Y may not be a0-a3 for the *dsp()* special function.

II.IV Addressing Modes

DSP3210 assembly language exhibits a syntax very similar to 'C'. The notation conventions are as follows: a0-a3 are the accumulators (DAU), and r0-r22 are the CAU registers. Instructions take the following appearance:

```
r2 = (long)r1      ; CAU register direct: store the contents of r1 in r1
r1 = (long)*r1      ; store value pointed to by r1 in r1
r1 = (long)r1 + 1    ; increment r1 by 1
*r2++ = (long)r1     ; postmodify increment r2 after storing r1 there (in *r2)
r3 = (long)r1 + r2    ; add two numbers in r1, r2: store the result in r3
r3 = (long)*r1++r2    ; post modify increment r1 by r2: store the result in r3
a2 = a2 + *r2 * a3    ; use that pipeline!
```

The following table lists the various addressing modes supported by the DSP3210:

Addressing Mode	Instruction Type			
	CA Data Move Group (CAU Reg)	CA Data Move Group (I/O Reg)	CA Arithmetic/ Logic Group	DA M/A & Special Func
Short Immediate	Yes			
24-Bit Immediate	Yes			
Memory Indirect	Yes			
CAU Register Direct	Yes	Yes	Yes	
IO Register Direct	Yes			
DAU Register Direct				Yes
Register Indirect	Yes	Yes		Yes
Register Indirect with Postmodification	Yes	Yes		Yes

II.V Latency Issues

The most difficult aspect of programming the DSP3210 is being aware of latency in the instruction pipeline. There are four cases in the DAU when pipeline affects latency. The cases are:

1. *DA Memory Writes*. When a DA instruction specifies a write to memory, the value written is not available to be read from that location until four instructions later (a three instruction latency). For example:

```
*r3 = a0 = a0      ; Instruction 1
*r3 = a3 = a3      ; Instruction 2
.                  ; Instruction 3
.                  ; Instruction 4
a1 = *r3            ; Instruction 5
```

The value read in instruction 5 is the value written in instruction 1, not instruction 2. Instructions 3 and 4 are latent instructions for instruction 1 and instructions 3, 4, and 5 are latent for instruction 2.

2. Accumulator as Multiplier Input. When an accumulator is used as an input to the multiplier, its value is established no sooner than three instructions prior to the multiply instruction (a two instruction latency). Note that this also applies to an accumulator using the X field of an instruction of the form:

$$[Z=] aN = [-]Y \{+, -\}X$$

For example:

```
a0 = a0 + *r1**r2      ; Instruction 1
a0 = a0 + a1           ; Instruction 2
.                     ; Instruction 3
a2 = a0 * a0           ; Instruction 4
a1 = a2 + a0           ; Instruction 5
```

The value of a0 used in instruction 4 is calculated in instruction 1. The value of a0 used in instruction 5 is calculated in instruction 4 since there is no latency effect on accumulators used as inputs to the adder.

3. Branching. When a CA Control Group instruction of the form *if()*goto, *call*, *return*, *goto* is executed, the instruction immediately following is also executed before the branch occurs. This is commonly referred to as a delayed branch. The *ireturn* instruction is different, and execution of the base-level program resumes in the following instruction cycle. For example:

```
if(eq) goto over       ; Instruction 1
r1 = 3                 ; Instruction 2
.                     ; Instruction 3
```

Instruction 2 is executed even if the condition is true and the branch is taken. If this is undesirable, a *nop* can be placed after the branch instruction, or if possible, the instructions can be rearranged. Because of this latency, a complex situation arises if successive branch instructions are coded in the following manner:

```
goto A                 ; Instruction 1
goto B                 ; Instruction 2

A:
.
B:
.
C:
```

The order of execution is instruction 1, instruction 2, A, B. If the instruction at A is not a *goto*, execution continues from B. If the instruction at A is *goto* C, the order of execution is instruction 1, instruction 2, A, B, C, and execution continues from C. Successive branch instructions are useful in some applications.

4. Conditional Branching on DAU Conditions. A DAU condition tested by a conditional branch or conditional arithmetic/logic instruction is established by the last DA instruction that affects DAU flags no sooner than four instruction prior to the test (a three instruction latency):

```
a0 = a0 + a1           ; Instruction 1
a2 = a0 * a2           ; Instruction 2
.                     ; Instruction 3
.                     ; Instruction 4
if(agt) goto next      ; Instruction 5
```

; Instruction 6 (latent instruction)

The condition tested in instruction 5 is established by instruction 1, not instruction 2. Because of this latency effect, use the zero-latency *ifalt()*, *ifagt()*, and *ifaeq()* functions where possible (see DA Special Instructions). The DA condition tested by these conditional accumulator loads is established by the last DA instruction that affected the DAU flags.

II.VI Amiga Hardware Integration

The DSP3210 is integrated into the Amiga system as a bus-master device. It shares the system bus with the CPU. Most of what the DSP does is dependent on real-time response; it must be guaranteed a minimum response time when it requests the system bus. The DSP is given priority over the CPU when requesting the system bus. The DSP sees the entire 32-bit address range that the CPU sees, and all the caveats of programming the Amiga apply to the DSP. If a program runs awry, it can easily trash the system. The DSP can also interrupt the CPU using either a level 2 or level 6 interrupt, and the CPU can interrupt the DSP using the DSP INT1 signal. The CPU only interrupts the DSP to bootstrap it the first time the system software is loaded.

The DSP3210 can be connected to one or more I/O devices on its serial port, but only one device can be active at any given time. Both a 16-bit stereo audio CODEC and telephone CODEC are possibilities. VCOS Device drivers will be provided for both of these devices, and information on writing drivers for other devices will be available from either Commodore or AT&T.

III. System Software

III.I Overview

A DSP is a very sophisticated piece of hardware, but it is quite useless without good software. In designing the system software for the DSP3210, AT&T set out to meet the following goals:

- *Provide maximum utilization of the silicon resource with minimum system overhead.*

A DSP can add significant cost to a system. If it can provide only one function or one function at a time, then it becomes difficult to justify its added cost. The DSP should function well at a variety of tasks and be able to execute them simultaneously.

- *Provide a consistent and straightforward DSP programming interface.*

DSPs can be difficult to program. To help alleviate this difficulty, standard tools and support libraries as well as a flexible model for performing I/O and communicating with DSP tasks should be provided.

- *Provide complex functions as a standard part of the system.*

There are relatively few good DSP algorithm programmers; there are many good application programmers. This fact must be exploited in order for a DSP subsystem to be successful. Functions for a broad range of popular DSP applications should be provided with the system.

The software support for the DSP3210 consists of two pieces: the DSP3210 Software Development Tools and the VCOS Operating System. The development tools allow application programmers to write and debug DSP3210 applications in both C and assembly language. VCOS provides a real-time multitasking environment, as well as a library of multimedia algorithms for the DSP3210. Application development involves selecting from a suite of algorithms which have already been implemented and are required by the application, or designing and implementing the DSP algorithm from scratch. Then the interface between the host application and the DSP is designed and implemented.

III.II VCOS

III.II.I Overview

VCOS provides a platform-independent environment for multitasking applications on one or more DSPs. VCOS consists of six key components:

- **VCOS** The VCOS Kernel which runs on the DSP3210. VCOS runs on the DSP and has one very basic function. It traverses an execution list and executes DSP code modules in turn at a pre-determined frame rate (the default is 10ms). There are two execution lists, BLEVEL and ILEVEL (background and interrupt levels). The BLEVEL list is executed round-robin while the ILEVEL is executed once per external interrupt. VCOS also provides caching control and buffer management.
- **VCAS** The VCOS Application Server. VCAS provides routines for the host system (the Amiga) to load, execute and communicate with DSP tasks.

- **VCRM¹** The VCOS Resource Manager. VCRM provides low-level resource management for the DSP. It manages the allocation of devices attached to the DSP, such as speakers, microphones, and telephone lines. It also manages run-time load balancing of tasks on multi-DSP system configurations.
- **VCSIM¹** The VCOS Simulator. VCSIM provides a full non real-time simulation of the VCOS environment. VCOS tasks may be run and debugged on systems without DSP hardware.
- **VCD** The VCOS Debugger. VCD provides a full symbolic debugging environment for DSP applications running under VCOS. VCD supports real-time execution using both real-time sampled data and disk based data. VCD also features breakpoints and single step facilities.
- **MML** The VCOS Multimedia Module Library. The MML is a collection of basic and advanced signal processing algorithms implemented as VCOS modules. MML modules may be combined with other MML modules or with user created modules to create complex processing streams on the DSP. The VCOS MML consists of the following modules as of version 1.0:

Integer sample rate converter (1->2, 1->3, 2->1, 3->1)
 16/24Kbps subband coder
 G.722 7Khz speech coder
 DTMF generator/detector
 JPEG still image coder and decoder
 MPEG level 2 audio coder and decoder

There are other modules that will be available in the near future:

Call progress detector²
 Non-integer sample rate converter
 4.7Kbps CELP coder
 Delta-Cepstrum feature extractor
 Text to phones, phones to LPC, LPC to speech
 Speaker trained, isolated word recognizer
 Speaker independent connected digit recognizer
 Talker verification
 3D graphics library²
 MIDI music synthesizer with EMU Proteus sound libraries²
 Perceptual image coder
 Perceptual audio coder
 V.22bis MNP5 Modem (V.22bis, Bell 212A, V.23, V.21, Bell 103)²
 V.32 modem with fallback²
 V.29 G3 Fax modem with fallback²

II.II.II Visible Caching

A VCOS application is comprised of two parts: the DSP application or algorithm and the host application. The DSP application is comprised of code "Modules" which can be combined to

¹ VCRM and VCSIM are part of the VCOS 1.1 release and have not been ported to the Amiga at the time this document was created.

² These components and modules are included in the AT&T 1.1 release

form a "Task". The host application communicates control and parameter information to and from the DSP task. Amiga application programmers can view and utilize these DSP tasks very much like native Amiga subtasks. Loading and preparation of DSP code is managed by the host. Code sections may be swapped in or out of on-chip memory on demand as DSP tasks are executed. There are three distinct address spaces under the VCOS DSP/Host model:

Host address	Host memory (physical, contiguous by section, locked)
DSP address	DSP physical address (mapped into host memory)
Execute address	Actual execute location (host or on-chip memory)

Modules are first loaded into host memory by the VCOS loader. A module may consist of several relocatable sections. Any or all of these sections may be cached on chip (as long as they fit in the available memory space) at the discretion of the application programmer. A module may, therefore, be represented in all three address spaces at any given time, for example:

1. Host loads module code sections into its memory (shared by DSP)
2. Host "downloads" code to DSP memory (on a bus-master just a translation or relocation)
3. Module code is cached by the DSP and is executed from on-chip memory.

In effect, all caching mechanisms are visible to the programmer, hence the term "Visible Caching". Tasks and their respective modules are executed on one of two execution lists: background or interrupt (also called BLevel and ILevel). The background list is executed in round-robin fashion (non-preemptive). The interrupt list is executed once per external interrupt and is used for time-critical I/O tasks (the code is guaranteed to be executed when an interrupt occurs). Deterministic execution is achieved through frame-based processing. Each DSP task is designed to process a block of data ("samples") in a given amount of time. The combination of visible caching and block-processing maximizes performance through optimal use of on-chip memory, and minimizes impact on host bus bandwidth by processing multiple samples per frame. Modules may save state information before exiting and load state information at execution time. Data is passed between modules and between modules and the host using "buffers". There are three types of buffers under VCOS:

AIAO	All In All Out (frame synchronous, cacheable, random access)
FIFO	First In First Out (asynchronous)
PARAM	Parameter (local shared, random access)

AIAO buffers are typically used for critical real-time DSP I/O. These buffers are static and can reside in on-chip memory (cached). AIAOs manage a fixed size data stream, and are serviced every frame. FIFO buffers are asynchronous in nature and may not get serviced every frame. FIFO buffers are used for managing sequentially accessed data streams (e.g. audio samples). PARAM buffers can contain any random access data required by the application and can be used to map host physical addresses into a DSP application. PARAM buffers have a fixed size, unlike AIAO and FIFO buffers which can have their size determined at load time. Both FIFO and PARAM buffers can be used for inter-Module and Module to Host communication. A typical application could be described by the diagram shown below in Figure 1.

VCOS also provides for special "Device Driver" Modules. These Modules are implemented to take advantage of system specific hardware (e.g., CODECs) and have standard calling interfaces. Once such a Module is written for a given device, any VCOS Module can take advantage of it for I/O. Currently, there is only one serial port, and therefore, only one active device on the DSP at any given time.

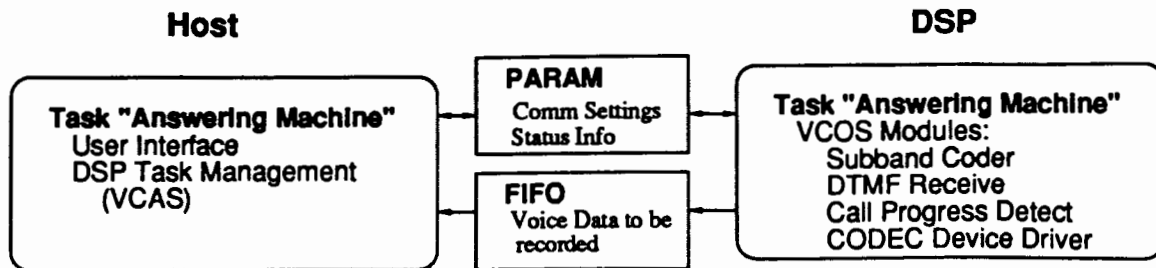


Figure 1: A Typical VCOS Application

II.II.III Sample Application

The following application loads and runs two tasks: an 8Khz device driver and a task which can feed raw 8-bit sample data to the driver. It then plays a selected data file through the task.

```

/* play.c
 *
 * Play Application:
 *   loads and runs iada8.tbd, then loads p_l8.tbd,
 *   then, following a pause, plays out data_file thru task p_l8
 */
#include <exec/types.h>
#include <exec/libraries.h>
#include <libraries/vcasbase.h>

#include <pragmas/vcas_pragmas.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/vcas_protos.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

void onbreak(void *);

struct VCASBase *VCASBase;

#define DSPid 1
#define BUFSIZE 0x7fffL /* buffer size for disk i/o */

char *dd_tbd = "iada8.tbd", *app_tbd = "p_l8.tbd", /* task/data file names */
      data_file[80];
int tidDD = -1, tidPlay = -1; /* task id */
int fd;

```

At this point all the required include files have been referenced: "clib/vcas_protos.h", which has all the function prototypes for VCAS as provided by AT&T, "pragmas/vcas_pragmas.h", which has all the compiler pragmas for the VCAS routines, and "libraries/vcasbase.h", which contains the structure definitions for the VCAS library base.

```

int kbhit(void)
{
    BPTR in;

    in = Input();

    return (WaitForChar(in, 1000) ? 1:0);
}

void cleanexit(char *msg)
{
    if (msg != NULL) printf("Exiting: %s", msg);
    if (tidDD && (vcStopTask(tidDD) < 0))
        printf("\nvcStopTask failed\n");
    if (tidDD && (vcDeleteTask(tidDD) < 0))

```

```

        printf("\nvcDeleteTask failed\n");
    if (fd >= 0) close(fd);
    if (VCASBase)
    {
        CloseLibrary((struct Library *)VCASBase);
        VCASBase = NULL;
    }
    exit(0);
}

int init(void)
{
    VCASBase = (struct VCASBase *)OpenLibrary("vcas.library", 0L);
    if (!VCASBase)
    {
        printf("couldn't open vcas.library\n");
        return (0);
    }

    if (vcAddTask(DSPid, dd_tbd, &tidDD) >= 0)
        if (vcStartTask(tidDD) >= 0)
            return (1);
        else
            printf("\nvcStartTask(%ld) failed\n", tidDD);
    else
        printf("\nvcAddTask(%s) failed\n", dd_tbd);

    return (0);
}

```

The above utility routines, *init()* and *cleanexit()* provide all that is necessary to initialize VCAS, load and start the device driver task and clean up when exiting. Note the version number for the shared library has not been determined at this time. Programming under VCAS is similar to programming for the Amiga in that all resource allocations must have corresponding deallocations.

```

int Play(long hf, int fd)
{
    long lReadCount, lWriteCount, *lpWrite;

    if (vcGetFifoWritePtr(hf, &lWriteCount, &lpWrite) < 0)
        return -1;
    if (lWriteCount != 0x0)
        printf("... Called vcGetFifoWritePtr(), lWriteCount: %ld \n", lWriteCount);
    if (lWriteCount > BUFSIZE)
        lWriteCount = BUFSIZE; /* limit moves */
    lReadCount = read(fd, (void*)lpWrite, (unsigned int)lWriteCount);
    printf("..... read only %ld bytes from file\n", lReadCount);

    if (lReadCount < lWriteCount)
    {
        long li;
        lReadCount &= ~0x3L; /* ensure word boundary */

        if (lReadCount == 0L) /* wait for dsp to empty FIFO */
        {
            do
            {
                if (vcGetFifoReadCount(hf, &li) < 0) return -2;
            } while (li != 0L && !kbhit());
            if (vcUpdateFifoWritePtr(hf, lReadCount) < 0) return -3;
            printf("... Called vcUpdateFifoWritePtr() \n");
            return 0;
        }
    }
    if (lReadCount != 0)
    {
        if (vcUpdateFifoWritePtr(hf, lReadCount) < 0) return -3;
        printf("... Called vcUpdateFifoWritePtr() \n");
    }

    return 1;
}

```

The above routine reads data directly from disk into the FIFO named "fta_in". It first checks to see how much room is available in the FIFO by calling *vcGetFifoWritePtr()*. It limits the size of

writes to a maximum buffer size that it will be reading from disk. Once it has read the data in, it updates the FIFO indices by calling `vcUpdateFifoWritePtr()`. If it first determines that it has reached end-of-file, it waits for the FIFO to empty by checking with `vcGetFifoReadCount()`. (Note that this busy-wait technique should actually be avoided in practice.) It is used here for the sake of brevity.

```
main( int argc, char** argv)
{
    static long hfPlay;                /* fifo handle */
    long li, lj;

    (void)onbreak(cleanexit);

    if (argc != 2) {
        printf("\nusage: play data_file"
               "\n data_file is the file to be played out (eg: sv.l8)"
               "\nexample: play sv.l8 \n");
        cleanexit(NULL);
    }

    strcpy(data_file, argv[3]);
    printf("dd_tbd: %s, app_tbd: %s\n", dd_tbd, app_tbd);

    if (init())
    {
        if (vcAddTask(DSPid, app_tbd, &tidPlay) < 0)
            cleanexit("\nvcAddTask failed\n");

        if (vcGetFifoHandle(tidPlay, "fta_in", &hfPlay) < 0)
            cleanexit("\nvcGetFifoHandle failed\n");

        /* open the data file */
        if ((fd = open( data_file, O_BINARY | O_RDONLY)) < 0)
            cleanexit("\ncan not open data_file %s\n", data_file);

        /* preload the output FIFO */
        if (Play(hfPlay, fd) < 0)
            cleanexit("\nERROR: Play()\n");

        if (vcStartTask(tidPlay) < 0)
            cleanexit("vcStartTask failed\n");

        /* play out the file */
        for (lj = 0L; ; lj++) {
            li = Play(hfPlay, fd);

            if (li > 0) {
                if ((lj&0x3ff) == 0) printf(".");
            } else if (li < 0) {
                printf("\nERROR: Play() returned %ld\n", li);
                cleanexit(NULL);
            } else break;
        }
        /*-- done --*/

        if (vcStopTask(tidPlay) < 0)
            cleanexit("\nvcStopTask failed\n");

        if (vcDeleteTask(tidPlay) < 0)
            cleanexit("\nvcDeleteTask failed\n");
    }
    cleanexit(NULL);
}
```

The main routine parses the data file name, adds the play task and opens the data file. It then gets a pointer to the input FIFO using `vcGetFifoHandle()` and fills it with a block of data by calling the `Play()` routine once. The play task is then started - note this technique: access to the task buffers is available before the task is actually started. The file is then played out, the play task is stopped, then removed and the program exits through the `cleanexit()` routine.

III.III Development Tools

A full suite of development tools is available for developing DSP and VCOS applications under AmigaDOS. The tools include:

d32ar	Archiver/Librarian
d32as	Assembler
d32cc	C Compiler
d32cpp	C Preprocessor
d32dump	Dump Section Information
d32ld	Link Editor
d32make	Maintain and Update Related Files
d32nm	Print Name List of Object Files
d32optim	Code Optimizer
d32sect	Relocatable Code Section Identifier
d32size	Print Section Size of Object Files
d32strip	Strip Symbol Information
d32trans	DSP32C Object Code Translator

There are two ways to develop DSP code for VCOS. One is to use the DSP3210 Assembler and the other is to use the DSP3210 C Compiler. The Assembler provides the optimum code speed and size for a DSP application. The C Compiler should only be used for non-critical code sections and for code which will not be distributed as a product. DSP applications are very time-critical. Currently, only by programming directly in assembly language can the optimum performance be achieved. The tools are organized in a directory structure as follows:

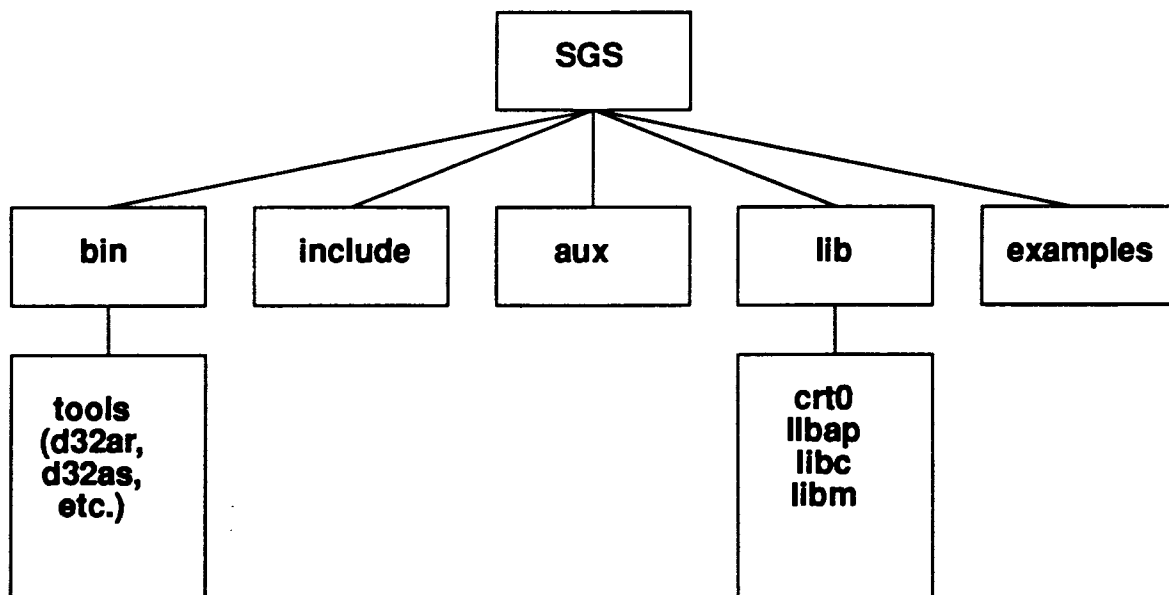


Figure 2: The DSP3210 Software Generation System (SGS)

The following environment variables are used by the tools:

DSP3210SL	The root (SGS) directory
DSP3210_AsmPP	Which preprocessor to use
DSP3210_Aux	Where the auxilliary files are

DSP3210_Includes
DSP3210_Temp
DSP3210_Tools

Where the include files are
Where to place temp files
Where the executable are

The auxilliary files include macros and helps files for the simulator and binary files containing boot code for the DSP3210. Refer to section V or the AT&T manuals for more information on using the tools for application development.

IV. Software Implementation Reference

IV.I Overview

The software implementation of VCOS for the Amiga is a flexible layered architecture. It is designed to allow easy expansion or change of the underlying hardware without the need to re-implement all the higher layers of software. The reference implementation of VCOS consists of the following items:

VE.LIB	Low level hardware functions
VC.LIB	Basic VCAS functions
VD.LIB	VCAS debugger support functions
VCD.LIB	Basic VCD functions

These are all originally designed as static/linkable modules. On the Amiga, VE and VC have been abstracted into three Amiga Operating System entities: the dsp3210.resource, the dsp3210.device, and the vcas.library. The functions for VD and VCD remain static libraries, but VCD is layered onto the Amiga architecture so that it makes calls to VC through the vcas.library:

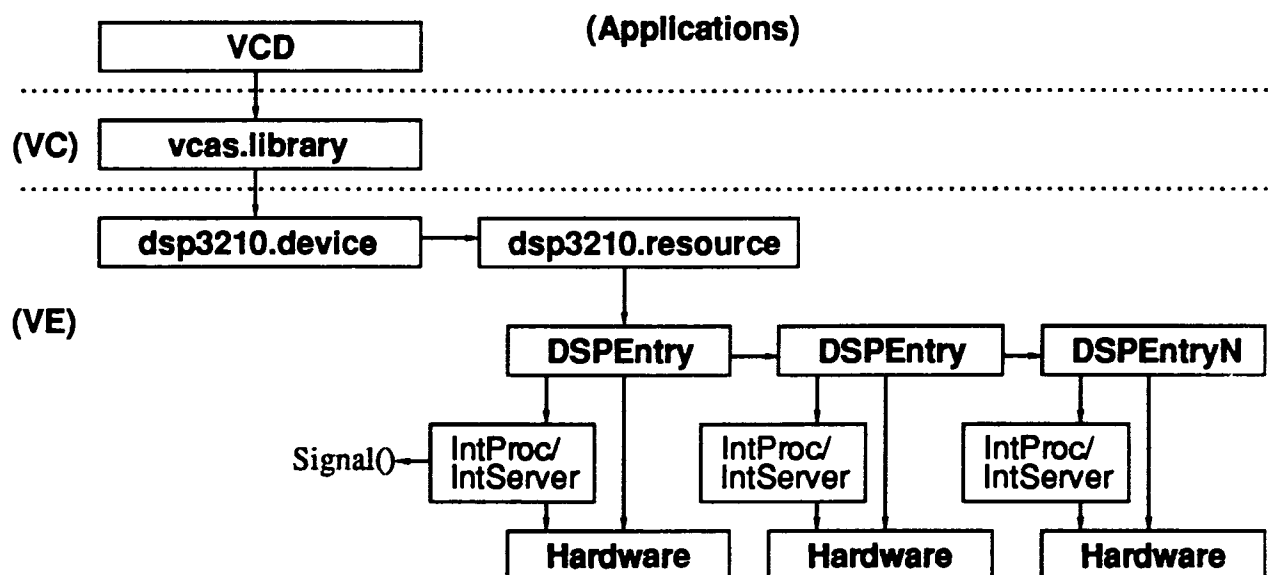


Figure 3: Amiga VCOS Software Architecture

These abstractions provide a clean and elegant system for supporting multiple DSPs as well as different hardware and software architectures in the system, including those which may be supplied by Commodore-Amiga, Inc., AT&T, or third party vendors. Most application writers will only use the VC interface of the software architecture (vcas.library). This documentation on how to access the lower layers has been provided so that people who wish to write their own DSP operating system or environment may do so in a system friendly and compatible manner.

IV.II. The dsp3210.resource

IV.II.I Overview

The resource provides hardware/vendor specific support for basic low-level functions required by

the system software. The dsp3210.resource appears as follows and is described in <resources/dsp.h>:

```

struct DSP3210Resource
{
    struct Node      dr_DSPNode;      /* Node to link into Exec. */
    struct MinList   dr_DSPList;      /* List of DSPs */
    UWORD            dr_NumDSP;        /* How many DSPs in system */
    UWORD            dr_Flags;         /* Special flags */
}

struct DSPEntry
{
    struct MinNode dsp_Node;           /* Link to next DSP */
    LONG (*DSPInit)();                 /* Function for Init */
    LONG (*DSPHalt)();                 /* Function for Halt */
    LONG (*DSPStart)();                /* Function for Start */
    LONG (*DSPRdMem)();                /* Function for Read */
    LONG (*DSPWrMem)();                /* Function for Write */
    LONG (*DSPRegInt)();               /* Function for Registering w/IntHandler */
    LONG (*DSPClrInt)();               /* Function for De-Registering w/IntHandler */
    struct Task *DSPIntr;              /* Pointer to the interrupt processor */
}

```

IV.II.II Resource Creation

A vendor will chain into the resource when their hardware is configured into the system. The vendor software should first check to see if the dsp3210.resource already exists in the system. If it does, they must chain their entries onto the existing DSP entries in the resource. If the resource does not exist, it is the vendor's responsibility to build one and then create the entries it needs. The following code fragment describes how to check for the resource and create one if it doesn't exist:

```

if (!(resource = OpenResource("dsp3210.resource")))
{
    Forbid();
    resource = AllocMem((long)sizeof(struct DSP3210Resource), MEMF_PUBLIC|MEMF_CLEAR);
    NewList((struct List *)&resource->dr_DSPList);
    resource->dr_DSPNode.ln_Type = NT_RESOURCE;
    resource->dr_DSPNode.ln_Pri = 0;
    resource->dr_NumDSP = 0;
    resource->dr_DSPNode.ln_Name = (char *) AllocMem(RESOURCE_NAME_LENGTH, MEMF_CLEAR);
    sprintf(resource->dr_DSPNode.ln_Name, "dsp3210.resource");

    AddResource(resource);

    /* For each DSP, do the following: */
    {
        entry = AllocMem((long)sizeof(struct DSPEntry), MEMF_PUBLIC | MEMF_CLEAR);

        /* Assign function vectors here */

        AddTail((struct List *)&resource->dr_DSPList, (struct Node *) entry);
        resource->dr_NumDSP++;
    }
    Permit();
}

```

The resource remains resident in memory until the system is rebooted. In addition, the resource creates a task to manage interrupts to and from all the DSPs it is responsible for (see section II.IV below). For a motherboard based DSP, the resource will either be located in ROM or created using an AutoConfig binding mechanism like BindDrivers.

IV.II.III Resource Function Vectors

The DSPEntry functions are re-entrant and are executed via the device driver, either in the caller's

context if quick I/O is used, or in the device driver's context if non quick I/O is used. Their function and parameters are as follows:

<u>Routine</u>	<u>Arguments</u>	<u>Description</u>
DSPInit	(void);	Initialize all DSPs
DSPHalt	(int dspNum);	Halt individual DSP
DSPStart	(int dspNum);	Start individual DSP
DSPRdMem	(int dspNum, ULONG from, APTR to, ULONG size);	Read from DSP memory
DSPWrMem	(int dspNum, ULONG to, APTR from, ULONG size);	Write to DSP memory
DSPRegInt	(int dspNum, int modID, BYTE signal);	Register with IntServer
DSPClrInt	(int dspNum, int Tid);	De-register with IntServer

These vectors manage all the DSPs associated with a single complete hardware entity (e.g., the motherboard, a single expansion board).

IV.II.IV Interrupt Management

The DSP3210 shares the system bus with the Amiga CPU and custom chips. The VCOS software, much like the Amiga Exec, was designed to help programmers write applications that cooperate and minimize impact on the host. Like the Amiga Exec, it is still possible to write applications which severely impact system performance, especially since the DSP is given the highest priority on the system bus. The minimal VCOS reference implementation was designed for a single-tasking system, MS-DOS. Under this implementation, synchronization between the DSP and the host occurs as follows:

Host tells DSP to do something or requires some event to complete on the DSP
Host busy-wait till the event condition is satisfied

So, if the host was waiting for the DSP to complete some function, it might check a PARAM buffer which was used as a synchronization flag as follows:

```
vcGetDspParamHandle(tid, "flags", param1);      /* Required for non bus-master systems */
while (!param1->flag) {
    vcUpdateDspParam(param1);
}
```

Amiga programmers know that on a multitasking system, this style of programming can use excessive amounts of CPU time and degrade the performance of the entire system. It would be preferable to have the host application relinquish the CPU and be signalled via some interrupt mechanism when the desired condition was satisfied. A mechanism for doing this under VCOS has been provided, however, the details of the host interface differs from platform to platform. A VCOS macro called *Signal()* is provided for DSP applications to interrupt the host. The *Signal()* macro passes the VCOS Module ID and an application specific 32-bit value to the host via a FIFO.

When the resource is first created, an interrupt server is also created to manage interrupts from one or more DSPs associated with the resource. The resource maintains function vectors so that host applications may register to receive notification of an interrupt from a particular DSP (and de-register before they exit). Host applications may not interrupt the DSP at will. This is necessary so that the DSP may continue to manage real-time events without loss of data. Each DSP may have a separate host interrupt handling mechanism and associated software processor. To use the interrupt feature, an Amiga application must first register with the interrupt server. To do so, the application calls the resource function vector *DSPRegInt()*. In the future, a function vector in the vcas.library may be provided to access this function. The application passes the

following arguments to the function:

int Dsp	Which DSP the module is running on.
int modID	The ID of the VCOS module which will cause the event.
BYTE signalnum	The signal to set when notifying the task.

Once registered, the Amiga application may use the Exec function *Wait()* to sleep until the signal is set. This brief example demonstrates proper usage of the interrupt server under VCOS:

```
vcAddTask(dspid, "sync", &tid);

signalnum = AllocSignal(-1);
DSPRegInt(dspid, tid, signalnum);

vcStartTask(tid);
signals = Wait(SIGBREAKF_CTRL_C | 1L << signalnum);

DSPClrInt(dspid, tid);      /* Must de-register before exiting */
FreeSignal(signalnum);

vcStopTask(tid);
vcDeleteTask(tid);
```

IV.III. The dsp3210.device

IV.III.I Overview

The device driver provides atomic access to low level hardware functions. It does not directly control any of the DSP hardware, but provides a standard Amiga EXEC I/O abstraction for higher level applications. All low-level abstraction is provided by the hardware resource.

IV.III.II Implementation

The device driver has functions which provide atomic access to the DSP hardware. These functions include standard Exec device management (e.g., *OpenDevice()*), and Exec I/O vectors which call functions in the proper vendor provided hardware resource to control the DSP. The I/O commands, flags, and errors are defined in `<devices/dsp.h>` and are as follows:

```
/*----- Commands -----*/
#define DSP_INIT      (CMD_NONSTD)
#define DSP_HALT      (CMD_NONSTD+1)
#define DSP_START     (CMD_NONSTD+2)
#define DSP_READ      (CMD_NONSTD+3)
#define DSP_WRITE     (CMD_NONSTD+4)
#define DSP_REGINTR   (CMD_NONSTD+5)

/*----- Error Returns -----*/
#define DSPErr_NoDSPs  1
#define DSPErr_NoUnit  2

/*----- Flags for OpenDevice -----*/
#define DSPB_EXCLUSIVE  1
#define DSPF_EXCLUSIVE (1<<0)
```

IV.III.III Calling The Device Driver

The device driver is opened like any standard Exec device. The only flag available to the application programmer during the open is "DSPF_EXCLUSIVE". If this flag is set, an attempt will be made to open the unit for exclusive access; if another application already has the unit open (e.g., VCAS), the open will fail. Once a unit is opened for exclusive access, all other attempts to

open the unit by other applications will fail until the unit is closed by the exclusive opener. The following is a brief example of how to properly open, close and perform I/O to the device:

```
// Simple example to open the dsp3210.device, do simple I/O and close
// Compile with SAS/C: lc -cfikmqw -L testdev.c
//

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/dsp.h>

struct MsgPort *Dport = NULL;
struct IOStdReq *Dreq = NULL;
BOOL DevOpen = FALSE;

void cleanexit(char *msg)
{
    if (msg) printf("Exiting: %s\n");
    if (DevOpen) {
        CloseDevice(Dreq);
        DevOpen = FALSE;
    }

    if (Dreq) {
        DeleteIORequest(Dreq);
        Dreq = NULL;
    }

    if (Dport) {
        DeleteMsgPort(Dport);
        Dport = NULL;
    }

    exit (0);
}

void main(int argc, char **argv)
{
    Dport = CreateMsgPort();
    if (!Dport) {
        cleanexit("init: couldn't create MsgPort");
    }

    Dreq = CreateIORequest(Dport, (long)sizeof(struct IOStdReq));
    if (!Dreq) {
        cleanexit("init: couldn't create IORequest");
    }

    if (OpenDevice("dsp3210.device", 0, Dreq, DSPF_EXCLUSIVE) != 0) {
        cleanexit("init: Error on OpenDevice()");
    }
    DevOpen = TRUE;

    Dreq->io_Command = DSP_INIT;
    Dreq->io_Flags = 0;

    DoIO(Dreq);

    if (Dreq->io_Error) {
        printf("An error occured in DSP_INIT: %ld\n", Dreq->io_Error);
    }

    cleanexit(NULL);
}
```

IV.IV. The vcas.library

IV.IV.I Overview

The VCAS (VCOS Application Server) shared library provides all high-level task management for the VCOS operating system running on the DSP3210. The philosophy behind VCAS is to

offload all non-time critical VCOS management from the DSP to the host. This allows the DSP to spend its valuable cycles on executing DSP functions. VCAS is defined by the MS-DOS/MS-Windows reference implementation which provides the following functions:

```
vcBootDsp(int DspN)
vcResetDsp(int DspN)
vcStartDsp(int DspN)
vcStopDsp(int DspN)

vcAddTask(int DspN, char *TaskName, int *Tid)
vcDeleteTask(int Tid)
vcStartTask(int Tid)
vcStopTask(int Tid)

vcGetFifoHandle(int Tid, Char *FifoName, long *hfHandle)
vcGetFifoReadCount(long hfHandle, long *ReadCount)
vcGetFifoWriteCount(long hfHandle, long *WriteCount)
vcReadFifo(long hfHandle, long Count, long *DestPtr)
vcWriteFifo(long hfHandle, long Count, long *SrcPtr)
vcFifoSize(long hfHandle, long FifoSize)
vcInitFifo(long hfHandle)
vcGetFifoReadPtr(long hfHandle, long *Count, long **RdPtr)
vcGetFifoWritePtr(long hfHandle, long *Count, long **WrPtr)
vcUpdateFifoWritePtr(long hfHandle, long Count)
vcUpdateFifoReadPtr(long hfHandle, long Count)

vcGetHostParamPtr(int Tid, char *ParamName, char **ParamPtr)
vcGetDspParamHandle(int Tid, char *ParamName, long *ParamHandle)
vcUpdateHostParam(long ParamHandle)
vcUpdateDspParam(long ParamHandle)
vcUpdateDspParamItem(long ParamHandle, long ByteOffset, long Count)
vcUpdateHostParamItem(long ParamHandle, long ByteOffset, long Count)
```

VCOS itself is a small, tightly coded kernel running on the DSP. DSP application programmers make use of the following macros to implement their applications and communicate with the host system:

```
NewModule(EntryName)
NewSection(SectionName, SectionType)
AppendSection(SectionName)
Push(rS)
Pop(rD)
GetSectionSize(SectionName)
SetEntryPoint(EntryPoint)

LoadSection(SectionName)
SaveSection(SectionName)
LoadRunSection(SectionName, EntryPoint)

WriteFifo(FifoSectionName, ByteCount, SourceAddress)
ReadFifo(FifoSectionName, ByteCount, DestinationAddress)
GetFifoSize(FifoSectionName)
GetFifoWriteCount(FifoSectionName)
GetFifoReadCount(FifoSectionName)

AddressPR(Label)
SetBaseSR(Label, rB)
SetBaseHSR(Label, rB)
AddressSR(Label, rB)

Signal(Value)
```

Documentation for all of the above functions may be found in the VCOS Technical Reference (or refer to section V for a DSP programming example). There are several special files which are created to define the structure of each VCOS application which the shared library must be able to read (specifically during AddTask). These are the VCOS Task Build Files (TBDs), and the VCOS Loader Files (LDFs). The VCOS application programmer must be aware of these file formats and know how to manipulate them. In the future, there may be calls added to VCAS which abstract the file formats into VCAS system calls.

IV.IV.II Implementation

Application programmers open and make calls directly to the VCAS library. When the library is first initialized, it determines how many DSPs are available in the system (using the dsp3210.resource). It then attempts to open each DSP for exclusive access (using dsp3210.device). Once all the DSPs have been allocated, the library initializes VCOS for all available DSPs. Currently, the library uses the default VCAS mechanism of searching the VCOS.CFG file to determine how many DSPs there are in the system rather than using the provided resource. The need for this file may be removed in a future version of the software.

IV.IV.III Calling The VCAS Shared Library

The VCAS shared library is opened like any standard Exec library. The library base symbol for the C compiler is "VCASBase". The following is a brief example of opening the VCAS shared library and making calls to VCAS:

```
// Simple example to open the vcas.library, call VCAS and close
// Compile with SAS/C: lc -cfikmqw -L testlib.c
//
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <libraries/vcasbase.h>
#include <pragmas/vcas_pragmas.h>

main()
{
    int tid1;

    if (VCASBase = OpenLibrary("vcas.library", 0L) == NULL) {
        printf("Couldn't open vcas.library\n");
        exit(0);
    }

    if (vcAddTask(1, "ata", &tid1)) { // Now make a VCAS function call
        printf("vcAddTask() failed\n");
    } else
        /* Additional VCAS calls may go here */
        vcDeleteTask(tid1);
}

if (VCASBase) CloseLibrary(VCASBase);
}
```

Refer to the VCOS Technical Reference Manual for information regarding .TBD and .LDF files for DSP Tasks.

V. Application Development

V.I Overview

While a C Compiler does exist for the DSP3210, the DSP3210 Assembler is the most efficient and reliable tool for developing VCOS applications. The Assembler is the only route to producing the most optimal code (both in performance and size), which can often be a critical issue in a multitasking DSP environment.

V.II Using The Tools

The following brief example will serve to demonstrate the basic requirements for writing a DSP application:

```
#include <macros.h>

NewModule(main3)
NewSection(main3, sPROG)
NewSection(cmain3, sPROG)
NewSection(piparm, sPARAM)

AppendSection(main3)
    *sp++ = r18

    LoadSection(cmain3)      /* Cache these sections */
    r3 = cmain3
    call r3(r18)              /* Run our cached code */
    nop

    sp = sp--
    r18 = *sp
    nop
    return(r18)
    nop

/* Here begins the actual main program */
AppendSection(cmain3)
emain:
    *sp++ = r18              /* Push r18 onto stack */

    !r4 = (long)var1
    r3 = (long)*r4
    r3 = (long)r3 + 1
    *r4 = (long)r3

exitemain:
    sp = sp--                /* Decrement stack pointer */
    r18 = *sp                /* Pull r18 off stack */

    return(r18)
    nop

/* A PARAM buffer for exchanging information with the host */
AppendSection(piparm)
var1:    long    1          /* Param variable 1 */
var2:    long    2          /* Param variable 2 */
```

To assemble the above example, use the following command from the CLI/Shell:

```
d32as -b big -l -i INCLUDE:dsp -o test.o test.s
```

-b big Directs the assembler to generate code for a DSP3210 in big-endian mode.
-l Directs the assembler to generate a full listing file.
-i Directs the assembler where to locate include files.
-o Directs the assembler where to place the resulting object file.

After assembling, the application must be linked using the linker as follows:

```
d32ld -map -r -o test test.o
```

-map	Directs the linker to generate a memory map to standard output.
-r	Directs the linker to keep relocation information in the resulting object file.
-o	Directs the linker where to place the resulting object file.

The VCOS application is then ready for loading using the VCAS *vcAddTask()* call, VCD, or VCSIM.

V.III Debugging techniques

Both VCD and VCSIM can be powerful tools for debugging VCOS applications. VCD can be used to step through applications running in real-time on the DSP. In VCD breakpoints can be set, registers and buffers examined, and code stepped through. With VCSIM, a full DSP3210 and VCOS environment are simulated in software. VCSIM combines the simulation capabilities of d32sim with the debugging features of VCD. This makes it possible to develop and debug applications without having DSP hardware available (albeit much more slowly). VCSIM also removes the hardware variable from the debugging equation, which means faulty hardware is less likely to be the root cause of a problem.

Aside from using VCD and VCSIM, there are other useful methods of debugging applications in real-time. A VCOS application is typically a "black box". It is difficult at best for a programmer to know what is really happening inside his application while it is executing. One good technique (especially on bus-master systems like the Amiga) for debugging a VCOS application is to use debugging variables or counters in PARAM or FIFO buffers within the DSP code. A lot of time and trouble can also be saved by first building an external reference model for the application in C or C++. It is much easier to try new ideas in the higher level language than to laboriously code changes or new routines in assembler which may not even work, let alone be used in the final application.

VI. References

1. "DSP3210 Digital Signal Processor. The Multimedia Solution", Information Manual. AT&T, September 1991 printing.
2. "VCOS Multimedia Development Kit", Technical Reference". AT&T Release 1.0, March 1992 printing.
3. DSP3210 Support Software Toolkit Manual, Release 1.3
4. DSP3210 Support Software Library Manual, Release 1.3

VII. Acknowledgements

The author would like to thank the following people for their help and invaluable contributions to this project:

From Commodore:
Jeff Porter
Randell Jessup
Dave Haynie
Debashis Pramanik

From AT&T:
Gary Murakami
John Lynch
Charlie Mera



MediaDevices

by Christian Ludwig

This document contains preliminary information

The "MediaDevices" OS extensions described in this document are currently a work in progress. The specifications of the system are subject to change.

This document does not constitute a guarantee that Commodore will implement or offer the "MediaDevices" system, as described herein or in any other form, for sale at any given time, place, cost, etc. Nor is this a guarantee that Commodore will ever release anything called "MediaDevices." This is simply the initial design and working name for a prototype software system.

Table of Contents

Purpose

Coming to terms

What's the point?

Background

Implementation Specification

Overview

- Hierarchy

- Prefs editor

 - Named Engines

 - Install Process

- Capabilities Inquiry

- Multi-Stream Synchronization

- Standardized Command Set

- Action Strings Command

Opening a MediaDevice

- Unit Number

- Device Opening Flags

Sending Commands

- Command Inquiry

- Synchronization

- Command Control Flags

- Global MediaDevice Commands

- Other MediaDevice Commands

Closing a MediaDevice

media.device

- media.device functions

- Binding IOMedias into groups

- Commanding groups

MediaTypes - The DataTypes Interface to MediaDevices

DataTypes for External MediaDevices

DataTypes for I/O MediaDevices

Issues

What's Missing?

Appendix A - Device to Driver interface

Appendix B - IOMedia Structure

Purpose

This document is meant to be a first step toward defining a 1990s grade multimedia interface system for the Amiga. The system is not written. It is not even completely specified. As such it is still very much open to comment and criticism. It is my intent that together, Commodore and Amiga developers can design and implement a system of multimedia OS extensions that will rival or exceed the best of what is available on other platforms.

Coming to terms

Like most things even remotely related to the term multimedia, there is room for confusion in all the buzzwords. By using only a single meaning for words which cover multiple ideas, I hope to eliminate at least some of the potential confusion.

Device

For the purpose of this document, the word device is defined strictly as a standard Amiga Exec ".device".

Command

The word command is defined strictly as a command passed to a device.

Engine

An engine is a piece of hardware or software that does something that a user wants done. A laserdisc player is an engine. An MPEG board is an engine. A genlock is an engine. A software MPEG player might be an engine.

Named Engine

A named engine is a software construct that describes a relationship between an actual engine, a driver, a device, and a (possibly) user-supplied "name" for the engine.

Action

An action is a discrete happening that a user has in mind. For example, play, pause, configure, and search are all actions.

Driver

The word driver refers to executable code which is loaded by a device in order to translate commands into an action suitable for a particular engine.

Method

The OOP (Object Oriented Programming) word for an action.

CAPS

A taglist of a named engine's capabilities.

What is the point?

How to control all this multimedia stuff? Stuff like laserdisc players, audio tracks on CD, DAT & MiniDisc, VCRs, software-controllable genlocks, video switchers, picture scanners, frame grabbers, audio digitizers, movie grabbers, and many more pieces of hardware and software. All the sort of things that people find themselves hooking up to their machines when they get into multimedia. Currently, controlling all these things can be a software nightmare. Each separate engine requires a different software interface (a library, at best) and each interface is completely different from all the others. Letting applications control this stuff is the job of MediaDevices.

Letting the user control the stuff is the job of MediaTypes. MediaTypes provides a standard set of DataTypes-compatible objects that can be used to trigger MediaDevices. Control panels, picture-in-picture windows, prefs controls, and other user-interface components for different engines can all be brought up by applications through the use of standard DataTypes calls and methods.

Now, once you've started controlling the devices, it would be valuable to be able to manipulate the data that they generate. Most "movie grabber" boards, for example, can be counted on to have completely different native data formats. It is not reasonable to expect that they will all directly save in some convenient form like CDXL. This is where DataTypes comes in. Data gathered by MediaDevices is meant to be "dealt with" by DataTypes.

These three pieces all tie together in a way that lets applications transparently handle future hardware and software engines. Applications that want to make lots of multimedia features available, and yet don't need complete application-level control over particular engines can simply support DataTypes (and the MediaTypes subset) and learn a few new methods. Applications that want to be more closely tied to hardware, can also bypass MediaTypes and talk directly to MediaDevices for maximum control of engines.

Background

The Amiga computer's OS has always had the exec "devices" system for the integration of asynchronous multi-threaded access to hardware and software engines. The system works well, especially where there is a strict set of commands to be implemented. trackdisk.device and serial.device are good examples. Many third party replacements have been written for serial.device, one for nearly every "extra-ports" board that's been developed. Most of these devices will work with any piece of software that knows how to talk to serial.device. Problems arise when there is no consistent command interface among related devices. Worse yet, some engines have libraries as their application-level interface. While libraries are the

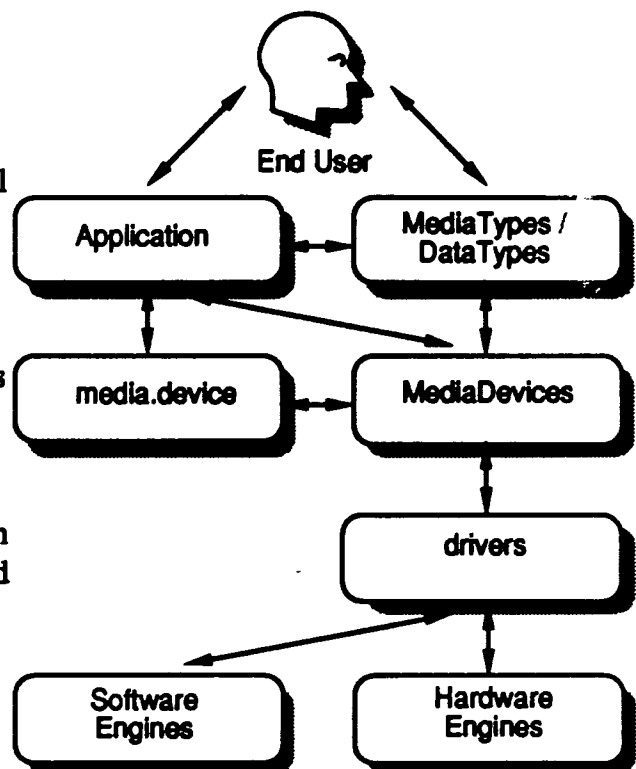
"easy way" to make functions available to applications, they do have their downside when it comes to engine control. Most importantly, libraries are synchronous by nature. This limits their usefulness as an engine control system. Certainly hardware which is capable of "taking care of itself" should (in most cases) be driven by devices, not libraries. Many pieces of multimedia hardware and software fall into this category.

MediaDevices is a specification for a system that tries to solve many of the multimedia engine control problems. It defines a stream of control information that is standardized across different engines of the same basic class, and even standardized across different basic classes of engines. It takes advantage of exec's devices system, allowing simple asynchronous access. By integrating realtime.library, we have also specified a standard system for the synchronization of MediaDevice-controlled engines to each other and to engines not controlled by MediaDevices.

Implementation Specification

Overview

Media.device acts as a controlling device, to which other devices can be bound by your application. Commands sent to media.device are automatically routed to all such bound devices. This provides a convenient means of starting multiple independent streams with just one command. Of course, messages have latency, and each device and/or driver takes a different amount of time to actually start doing what it's told, so applications can't depend on this system to truly synchronize engines. For true synchronization, a system such as realtime.library will need to be used at the driver level. Whether or not a driver supports various levels of synchronization (through realtime.library) is something that can be found out by checking a named engine's capabilities list.



Mmedia.device also includes several useful library-offset style functions (implemented in the same manner as **timer.device**'s functions) which can be used with the structures generated and used by other media devices.

A master device exists for each major class of multimedia hardware and software:

Devices and what they control

externaudio.device

For control of playback and recording on external audio devices. Examples: Redbook CD-Audio which can't be manipulated by the processor, audio tape devices, etc.

externmovie.device

For control of playback and recording on external video/audio devices. Examples include: Videodisc players and recorders, VCRs, and MPEG / Motion JPEG devices which playback video and/or audio signals directly (instead of creating a stream of images and audio which can be manipulated by the processor.)

videocontrol.device

For controlling video synchronization, overlay, and sizing devices. Examples: Genlock devices, picture-in-picture devices, software controllable proc-amps, etc.

audioio.device

For playback, recording, and building of digital audio. Examples: 8-bit parallel port audio samplers, digital audio cards, DSP devices which record and playback audio.

movieio.device

For playback, recording, and building of images or image sequences with or without associated synchronized audio. Examples: Motion JPEG boards which produce data in a form that is accessible to the CPU, flatbed image scanners, video frame grabbers, slide scanners, etc.

Separate drivers exist in a drawer for each type of device.

The driver system allows for the use of multiple simultaneous instances of each type of engine (through the use of multiple different drivers), and for multiple simultaneous instances of a particular engine (through the use of multiple identical drivers).

Prefs Editor

Named Engines

A prefs editor named *MediaDevice Links* allows the linking of an engine to particular devices and drivers. The editor allows a user to type in a human-readable name for the engine. It then asks which device should be used with that engine. Next, the user is shown the available drivers for that device. Once a driver is chosen, the prefs program opens the device with that driver, and asks the device to present a configuration requester. The human readable engine name, device name, driver name, and parameters returned by the configuration requester are then stored on disk. The human readable name for the engine is used as the name of an entry in a standard V39 utility.library name space. The special media.device command MDCMD_USERENGINE asks the device to present the list of available named engines, which the user can then choose from. Tags for the command allow the application to limit the list to particular devices, particular drivers, particular device or driver capabilities, or any combination of these.

Install Process

When a user installs a new engine, software that comes with the new engine can call the *MediaDevice Links* editor with command line options. These options point the prefs editor at a file which contains information to be added to the user's prefs area. In this manner, the user is saved the hassle of having to completely configure every new device that is added.

Capabilities Inquiry System

A capabilities inquiry system exists which allows drivers to report version number information, as well as a detailed tag-based capabilities list.

Multi-Stream Synchronization

A standard method for drivers to be linked to realtime.library conductors is provided. This system allows drivers to be synchronized to other drivers, DataTypes classes, and other applications, as well as external sync sources. The system also allows drivers to be the source of sync information for a conductor.

Standardized Command Set

An agreed upon set of standard commands is implemented for each root device.

Action String Command

One of these standard commands allows for an action string to be passed in, which is then parsed and converted to regular commands. Each command is then sent in turn to the corresponding driver. This allows programs to query users for an action string, and for easy implementation of ARexx interfaces to the devices.

Opening a MediaDevice

Unit Number

When opening a device, the unit number field has some special values which specify the driver to be used:

MDUNIT_ENGINENAME

This is the most useful and common way to open a MediaDevice. It will cause the device to examine the string pointed to by the `iom_EngineName` field of the passed-in `IOMedia` structure, which will then be searched for in the device's `EngineName` namespace. If found, the proper driver will be opened, and corresponding parameters loaded.

MDUNIT_USERENGINENAME

This unit type is similar to `MDUNIT_ENGINENAME`, except that instead of passing in an already filled `IOMedia->iom_EngineName` field, the device opens a window which lists the available named engines. Once the user has chosen an engine, the device opens the necessary driver, puts a pointer to the engine's name in the `IOMedia->iom_EngineName` field, and successfully returns. This unit is valuable when an application is only minimally aware of MediaDevices. By allowing the device to be opened in this manner, and then accepting strings from the user for use with the `MDCMD_ACTIONSTRING`, applications can still gain a large degree of multimedia functionality.

MDUNIT_DEFAULT

Means use the default named engine. The default for each device is specified by the user with the "MediaDevices Links" prefs editor. The parameters used are the defaults for the default named engine.

MDUNIT_NOCARE

Yields the first available named engine. This may be the default named engine, but will be another available named engine if the default is being used in exclusive access mode. The parameters used are the defaults for the available named engine.

Device opening flags

A group of special flags can also be specified at open time:

MDOPENFLAG_SHARED

This flag means that this device and driver combination (named engine) may be opened by other applications. This obviously implies that the driver may be opened multiple times. This may, however, be untrue of certain drivers. If a driver can only open in exclusive mode and has not yet been opened, a call to `OpenDevice()` with this flag will succeed, but subsequent attempts to open with that driver will fail until the driver is closed.

MDOPENFLAG_EXCLUSIVE

Is the opposite of `MDOPENFLAG_SHARED`. Exclusive access to the engine will be granted to the caller. No other program will get a successful `OpenDevice()` on this device and driver combination until you call `CloseDevice()`. Other callers can open the device, but only if they use a different driver. (By specifying a different named engine.)

MDOPENFLAG_USERVERBOSE

Allows the device to open windows and/or requesters if necessary to try to complete the open. This can happen if the engine is in exclusive use by another program, or if the driver can't be found. These windows allow the user to intervene, either by quitting the other program that is using the engine, or pointing the device at the proper driver (using an ASL file requester.)

MDOPENFLAG_DEBUG

Causes the device to report debugging information for each command that is sent to the device. This debugging information is sent via the `debug.lib` `kprintf()` function, and is only available if the debugging version of the device is used. The flag is ignored by the consumer distribution version of the device.

Additional flags may be defined for individual devices.

Sending Commands

Once the device is successfully opened, commands can be sent to it in the normal devices fashion, i.e., via an `IORequest` structure. The command itself is placed in `IOMedia->io_Command`, while any necessary data pointers and parameters go in their own special fields. Finally, the request is sent to the device via `SendIO()`, `DoIO()`, or `BeginIO()`.

Command Inquiry

The available commands are determined by the individual named engines (device / driver combinations). Whether or not a particular command is supported can be determined by using the capabilities inquiry system. Use the MDCMD_QUERYCAPS command with the IOMedia->IOData field pointing at a taglist of the commands that you wish to inquire about. The returned taglist will contain BOOL ti_Data values for each passed-in command.

Synchronization

If an engine supports external synchronization, the name of a RealTime conductor may be passed in IOMedia->io_ConductorName. The optional global command MDCMD_TAKESYNC allows the driver to synchronize to the conductor, while the optional global command MDCMD_MAKESYNC allows the driver to generate synchronization information for the conductor.

Command control flags (IOMedia->iom_MDFlags)

The following flags may be passed in IOMedia->iom_MDFlags, and control the execution of the particular command associated with the IO request:

MDFLAG_USERVERBOSE

This flag gives the device permission to open windows, if necessary, to allow the user an opportunity to intervene when common errors occur. Most errors that a driver reports will cause the device to put up simple "something's wrong, fix it. RETRY/CANCEL" requesters. Other errors may cause more elaborate windows to appear. Specifying this flag certainly does not guarantee that a command will succeed, but it should help. Without this flag, all driver errors will cause the command to fail.

MDFLAG_PREPARE

This flag instructs the driver to get ready to perform the command, but not actually do it. The command is acted upon to a particular point (that point being standardized for each command) and then pauses as if a CMD_STOP had been sent. What does this mean? Essentially, it helps applications properly synchronize the actions of multiple independent media streams. By sending various commands to different named engines, all with this flag set, one can later simply send CMD_START to all of the engines, causing them to start the intended action more or less simultaneously.

Global MediaDevice Commands

The following commands are to be supported by all MediaDevices.

CMD_CLEAR

Reset any internal buffers that the device may have

CMD_FLUSH

Purge all queued requests for the device (this does not affect active requests)

CMD_QUERY

Ask the driver to determine how many unprocessed bytes are available from the engine. Part of the method of last resort for talking to an engine.

CMD_READ

“Give me data” Straight through read. `io_Length` bytes of data are read directly from the engine (by way of the driver) and stored starting at the location pointed to by `io_Data`. Part of the method of last resort for talking to engines.

CMD_WRITE

“Have some data” Straight-through write. `io_Length` bytes of data pointed to by `io_Data` are sent directly to the engine (by the driver). Part of the method of last resort for talking to engines.

CMD_RESET

Reset the driver to its initialized state. All active and queued I/O requests will be aborted, and any memory dynamically allocated will be freed. The default parameters will be loaded.

CMD_START

Restart any paused I/O. This command will be used far more for MediaDevices than for traditional exec devices. See the definition of `MDFLAG_PREPARE`.

CMD_STOP

Pause all active I/O.

MDCMD_QUERYSTATUS

Returns a pointer to a taglist which describes the status of the device. The definition of this command will vary among devices. `STATUS` generally describes the state of any actions being performed by the driver, like “playing,” “stopped,” and conditions like “the door is ajar,” etc., as opposed to `PARAMS`, which describe user-configurable “adjustment values.”

MDCMD_QUERYCAPS

Returns a pointer to a taglist of “Capabilities.” These Capabilities will, of course, vary among devices and drivers. There are *device* capabilities, which the .device itself supports, *global* Capabilities, which all drivers must support,

and *driver* Capabilities, which drivers may optionally support. Boolean capabilities which are not included in the taglist are considered FALSE. There's a function in `media.device` that tells you if all the capabilities you want (which you store in another taglist) are in the CAPS taglist.

MDCMD_GETPARAMS

Returns a pointer to a taglist which describes the current state of the engine's user-configurable parameters. If a non-NULL pointer to a taglist is supplied, the same taglist pointer will be returned, but the list will have the new, updated values. Use `utility.library` to examine individual items in the returned taglist.

MDCMD_SETPARAMS

Takes a pointer to a taglist which describes a new set of parameters for the device and driver. Use `utility.library` to build taglists which will be applied to the individual items in the taglist. Items not appearing in your passed in taglist will be left alone.

MDCMD_GETPARAMSCONTEXT

This command will return a pointer to the engine's current user-configurable parameters. The length of the context is returned in `io_Actual`. This format of this data is undefined and self-contained. Applications must not try to interpret this data. The goal is to provide a means of saving parameters such that they can be reloaded later by the equivalent SET command.

MDCMD_SETPARAMETERCONTEXT

This command accepts a pointer to a self-contained block of data which represents the state of the engine. The data is checked for sanity by the device, and passed on to the driver. Applications must not try to fake-up this data.

MDCMD_USERABOUT

Asks the device to put up an about requester. To do this, the device asks the driver for a taglist of items to be displayed. Standard tagitems would be a pointer to the driver's name, a pointer to an image, etc. If the driver is incapable of passing back the taglist, the `.device` will display a simple window that names the current engine and driver, and displays version info. It's silly to `SendIO()` this one. You pass a screen pointer in `Param1`, or null for the default public screen. If the driver was performing some asynchronous action, the driver may or may not continue the action while the window is open, depending upon whether or not it has the `DYNAMIC_PREFS` capability. If possible, "About..." windows are made "frontdrop" windows, which can never be moved behind other windows.

MDCMD_USERPARAMS

Asks the device to put up a configuration requester. To do this, the device asks the driver for a taglist of items to be displayed. After the user sets the

parameters, they are activated (as if an `MDCMD_SETPARAMS` had been sent) and returned in the same format as `MDCMD_GETPARAMS`. If the driver is incapable of putting up the requester, the .device will display a simple window which explains that the driver has no user-adjustable parameters. Param1 should contain a screen pointer, or null for the default public screen. Param2 lets you specify which prefs requester comes up. Specifying `PREFS_MAIN` causes the main prefs window, if available (if the driver has the `PREFS_MAIN` CAP), to be displayed. All other prefs windows should be accessible from the main prefs window. Different devices and drivers have different groups of requesters, check the CAPS of the driver to find out what's available. It's possible that a particular driver may be able to continue performing a previously started asynchronous action while the prefs window is being displayed. If this is the case, it will report `DYNAMIC_PREFS` in its CAPS list, and is required to "demonstrate" the effects of prefs changes whenever possible. For example, a genlock driver which has `DYNAMIC_PREFS` and `ADJUST_HUE` CAPS is required to be able to show the hue being adjusted while the genlock is active and the user is twiddling the hue value.

MDCMD_ACTIONSTRING

Causes the device to parse a string pointed to by the `io_Data` field. The string consists of a series of standard device actions, each of which will be sent in turn to the driver as commands. The command does not reply until all of the actions in the string have been performed (or have errored out.)

MDCMD_TAKESYNC

This command asks the driver to sync to the RealTime conductor named in `IOMedia->ConductorName`. The driver will need to call `realtime.library` to create a player, and should link in a player hook if the driver wants time in some format other than RealTime heartbeat pulses. The driver is then responsible for synchronizing its commands to the time metered out by the conductor, including pausing when the clock pauses, etc.

MDCMD_NOTAKESYNC

The driver should cease accepting sync information from the named conductor.

MDCMD_MAKESYNC

This command asks the driver to be the external sync source for the RealTime conductor named in `IOMedia->ConductorName`. The driver will need to call the `realtime.library` to create a player, link the player to the named conductor, set the conductor to external sync mode, and then call RealTime's `ExternalSync()` function each time an external sync pulse happens.

MDCMD_NOMAKESYNC

The driver should cease generating sync information and release external sync on the named conductor.

Other MediaDevice Commands

Each class of MediaDevices has its own set of required, optional, and driver-specific commands. The required and optional commands for each class are printed as appendices to this specification.

Closing a MediaDevice

MediaDevices are closed with the standard exec CloseDevice() call. The root device takes care of closing the driver, flushing the driver if necessary, and deallocating any resources. Drivers are only flushed if the device's expunge routine is called and the opencount for a driver is zero.

MediaTypes - The DataTypes control interface to MediaDevices

A group of specialized DataTypes root classes provides a standardized user-interface to the MediaDevices system. These root classes allow users to objectify things like video scenes, audio snippets, and commands for video-control devices like genlocks or video scaling devices.

Why MediaTypes instead of DataTypes? The straightforward answer is that MediaTypes aren't real DataTypes. They are called and controlled in the same way, but they do not provide the same functionality.

Each MediaType has render methods that display "scenes," "transport controllers," and "preferences controllers." There is also a write method for scenes that has an IFF representation, allowing scenes, etc., to be stored as files or clipboard data.

Scene information includes an engine name, media identifier, user-supplied names and comments, and an event command string. The main trigger method for a MediaType causes the scene's command string to be sent to the corresponding device.

DataTypes for I/O engines

It is not reasonable (nor a particularly useful division of labor) to expect that drivers will directly generate data in the root format of a particular DataType. For example, MediaDevice drivers for a flatbed scanner should be concerned with properly and most faithfully reproducing the image data that the scanner generates. It is not necessarily the job of the driver to convert that data into a familiar form like ILBM. This is a job best left for

DataTypes, which has data conversion as one of its goals. To this end, an important part of any complete **MediaDevices** driver implementation (for I/O engines at least) will include a **DataType** class (or classes) for conversion to and from the native data format of the engine.

In cases where the **DataType** for the data generated by an engine requires access to the engine in order to properly perform a conversion or render, the **DataType** should call the proper device/driver combination, not talk to the engine directly. A good example would be a movie **DataType** for a mythical JPEG board. The job of the **DataType** is to translate from the board's native format (streams of JPEG pictures, most likely) to the root format of the movie class, and back again. If the **DataType** needs access to the board in order to do this (highly likely in this case) it should call the appropriate device/driver combination as necessary.

Issues

As with any evolving specification, issues of implementation and design remain unresolved. Below are the major ones to be considered.

- ☐ There is certainly a degree of overlap in the current spread of parent devices. There are valid arguments for having only one device, with all per-engine functionality coming from drivers. This would require that the drivers be larger, but would prevent the confusion that can result when a particular product actually performs the job of several engines.
- ☐ There are also many valid arguments for dividing up the devices in different ways. Combine the external devices, combine the I/O devices, fold `videocontrol.device` into one of the others, define an `audiocontrol.device`, etc., ad infinitum. All of these ideas (and many more) should be weighed carefully.
- ☐ Many of those who have already looked at this spec have remarked that, "In most cases, devices shouldn't bring up their own windows." This is probably true, for a number of reasons, so this is certainly an issue. Some of the most plausible options seem to be:
 1. Callers pass taglists to devices, which in turn pass them to drivers which then actually bring up the windows themselves. This is probably the worst solution, as it will inevitably lead to many different (harder to recognize) solutions to the same basic problems. Much like file requesters before ASL.
 2. The spec as it stands... Callers and drivers pass taglists to devices, which in turn bring up the window wherever the caller asks for it. This is similar to the way ASL currently works.

3. Drivers get taglists from devices, and build up a BOOPSI image sub-class on the fly. This object can then be placed wherever the calling application wants it. This is similar to what is planned for ASL.
 4. Drivers have BOOPSI stuff coded into them, which they return to the device and caller. This is similar to what is planned for ASL.
- ❑ There is some question as to the purpose and/or need for media.device. The goal of media.device was to have a convenient place to hang functions which would be useful to all users of MediaDevices, and to allow for the nearly simultaneous starting and stopping of multiple devices by binding together IOMedias, which would all be called in a big loop when a message bound for the group was sent to media.device.
 - ❑ There could of course, exist a media.library, or some such, which would supply most of media.device's functionality, but this would require that applications loop through multiple IOMedias themselves when they wanted to start or stop multiple ones at the same time. Obviously there will always be latency and overhead with the media.device approach.
 - ❑ The spec mentions compatibility with V37 of the OS, but actually makes heavy use of V39's name space management functions. How to address the name space issues is not covered here, and a decision needs to be made regarding whether or not V37 compatibility is necessary for these systems.
 - ❑ There has been considerable discussion with regard to whether or not drivers should be required to be able to deliver data in some "root" form. The spec as it stands allows drivers to only support the native format of the engine's data, which would then require manipulation by a corresponding DataType to be converted to some standard format. It has been argued that this is too much work for an application to do, and that some system should be designed whereby the data would be automatically converted by the driver. Perhaps all drivers would be required to be able to pass their native data to their corresponding DataType *before* the data is returned to the caller.

What's Missing?

This spec is missing major sections, most of them concerned with the command interface for particular types of engines. The absence of these sections has a great deal to do with the absence of these sorts of products from the Amiga marketplace. It is difficult to specify a reasonable minimum set of commands for a particular class of engine when there are no (or

very few) engines of that type available. In my opinion, this is a mixed blessing. Obviously the Amiga market would benefit from having certain of these types of products. On the other hand, there late entry does give us a unique opportunity to get this system (MediaDevices) close to "right" the first time around. I welcome commentary, criticism, ideas and suggestions. Just call, write, email, beat drums (if you think I'll hear), or whatever it takes.

This spec includes only a limited description of what the device to driver interface will look like. The essentials are presented for discussion.

The spec needs work in the area of media synchronization. The current definitions of commands like MDCMD_TAKESYNC and MDCMD_MAKESYNC are too loose, and will require stricter language to properly keep drivers synched to one another and other applications.

Appendix A

MediaDevice Device to Driver Interface Specification

Drivers for MediaDevices are implemented as standard Amiga shared libraries. All drivers are required to be fully re-entrant. Aside from the usual required library entry points (Open(), Close(), Expunge(), etc.), root devices expect that these libraries will have another standardized set of entry points, listed below. Each driver maintains an internal taglist of capabilities. Drivers which report that they can be opened in shared mode (MDCAP_SHARED) are required to keep special context info in the IOMedia structure. Each call to a standard function requires a pointer to an IOMedia structure. The io_EngineParamContext field is used to store multiple parameter contexts for different engines. As the driver executes a given command, it should fetch needed parameters from this context.

A separate command context is required for each IOMedia. This is guaranteed by the required use of a separate IOMedia for each caller. Similarly, a separate parameter context is required for each engine, and should be set up whenever the root device calls DOMDCommand with the MDCMD_SETPARAMETERCONTEXT command. This happens automatically for each open.

For all functions, errors should be stored in IOMedia->io_Error, and duplicated in the return value. Zero means no error.

Required entry points:

```
error = DoMDCommand(struct IOMedia *)
```

This function is the main entry point for a driver. Most device level commands are routed to a driver with this function. Drivers operating in shared mode use the context which is represented by unique IOMedia structures and pointers to engine contexts (put into the IOMedia structure by the driver at open time) to keep track of multiple threads.

In general, this function will consist of whatever context switching is necessary (none should be, if all context sensitive information is stored in the I/O request block), followed by a nice big switch statement that covers the required global commands and any additional device or driver specific commands.

```
error = MDOpen(struct IOMedia *)
```

Called when an application calls OpenDevice(). Gives the driver a chance to do any necessary allocations. No actual initialization of the driven engine should occur here, as the controlling device will call DOMDCommand() with the global MDCMD_SETPARAMETERCONTEXT and CMD_RESET commands.

```
error = MDClose(struct IOMedia *)
```

Called when an application calls CloseDevice(). At this point the driver is required to free resources associated with a particular IOMedia. No attempt should be made by a driver to keep track of the number of openers.

```
error = MDShutdown(struct IOMedia *)
```

Called at shutdown time. This allows the driver to deallocate all resources that it is using. Memory should be freed, sub-devices closed, ports released, etc.

Appendix B

IOMedia Structure

The IOMedia structure is an extension of the existing IOExt structure. This structure includes room for a number of application-supplied parameters per command, as well as a place for the devices and drivers to store context information needed for realtime.library synchronization and engine parameter storage.

```
struct IOMedia {
    struct IOStdReq IOMD;
/* STRUCT MsgNode
* 0  APTR  Succ
* 4  APTR  Pred
* 8  UBYTE Type
* 9  UBYTE Pri
* A  APTR  Name
* E  APTR  ReplyPort
* 12 UWORD MNLength
*  STRUCT IOExt
* 14 APTR  io_Device
* 18 APTR  io_Unit
* 1C UWORD io_Command
* 1E UBYTE io_Flags
* 1F UBYTE io_Error
*  STRUCT IOStdExt
* 20 ULONG io_Actual
* 24 ULONG io_Length
* 28 APTR  io_Data
* 2C ULONG io_Offset
*
* 30 */

    APTR iom_EngineName;
    APTR iom_EngineParamContext;
    ULONG iom_Param1;
    ULONG iom_Param2;
    ULONG iom_Param3;
    ULONG iom_Param4;
    ULONG iom_MDFlags;
    UWORD iom_Status;
    APTR iom_ConductorName; /* NULL-terminated string * for RealTime */
    APTR iom_PlayerInfo; /* A pointer to a PlayerInfo, filled by driver, for driver's use */
};
```

◆



MPEG and the Amiga/CDTV

by Jeff Porter

Introduction: What is MPEG?

MPEG stands for Motion Picture Experts Group, and is an international standard for audio and video compression, decompression, and synchronization that is optimized for CD-ROM data rates (MPEG1) sanctioned by ISO/IEC JTC1/SC29/WG11 and in the United States by ANSI X3L3 as standard number ISO11172.

The MPEG1 standard uses a picture size called Source Input Format (SIF) which for NTSC is 352x240 at 30 frames per second and for PAL is 352x288 at 25 frames per second. Since for many computer systems this is not full screen, most MPEG decoder chips double the horizontal pixels and double the lines for interlacing to give 704x480 at 60fps for NTSC and 704x576 at 50fps for PAL. The normal bit rate for MPEG video is about 1.1Mbps.

The audio coding technique uses a psychoacoustical model of the ear as well as coding and quantization to remove the sound that the human ear can't hear. This technique is similar to what Sony is supporting for the new MiniDisc and what Philips is supporting for DCC. Digital Broadcast Radio plans to use similar techniques, as well as one of the proposed HDTV standards. The bit rates for MPEG audio are between 32Kbps to 192Kbps per channel and the standard support three different Layers. Layers 1 and 2 are most common and quite similar. Both can achieve CD-quality sound, with layer 2 achieving this at slightly lower bit rates. Layer 3 was an attempt to keep the same quality of Layers 1 & 2 at half the bit rate, but most companies have stopped at Layer 2 due to the complexity of implementing layer 3.

With MPEG Video normally taking 1.1Mbps and Layer 2 Stereo at 192Kbps/channel, you have a total of 1.484Mbps, which is less than the 1.5Mbps of a standard CD-ROM. It is allowable to adjust the bit rate of the audio and video to suit each application and the MPEG decoder will compensate accordingly.

The list of companies contributing to the MPEG standard is quite vast. Nearly every computer company, consumer electronics company, hardware company, software company, and semiconductor company is a member (except Microsoft). The quality achieved with MPEG is quite impressive. The picture quality is much better than VHS in that it does not contain any static, herringbone, or other analog defects, and some say it rivals the quality of a laser disc. Occasionally, depending on the source material, you can notice some digital blocky defects, but generally, these are quite few. For a home consumer product and a desktop computer, the quality is much better than what you have today with QuickTime,

Video for Windows, AVM, and other proprietary compression standards. Commodore and our development community cannot by itself convince the movie and record producers to encode to their video to our platform without many large wheelbarrows full of money. By joining the standard, the videos will be coded once, and everyone can make use of them.

But What Is MPEG Good For?

MPEG can be used for movies. It will take two discs for a single movie, but so do large format laser discs. MPEG can also be used for music videos. Imagine if you will what is going to happen to the record industry with the advent of compression technology. Audio CDs of today will become MPEG video discs of tomorrow. The MiniDisc will replace the conventional audio only CD of today and cassette based Walkmans.

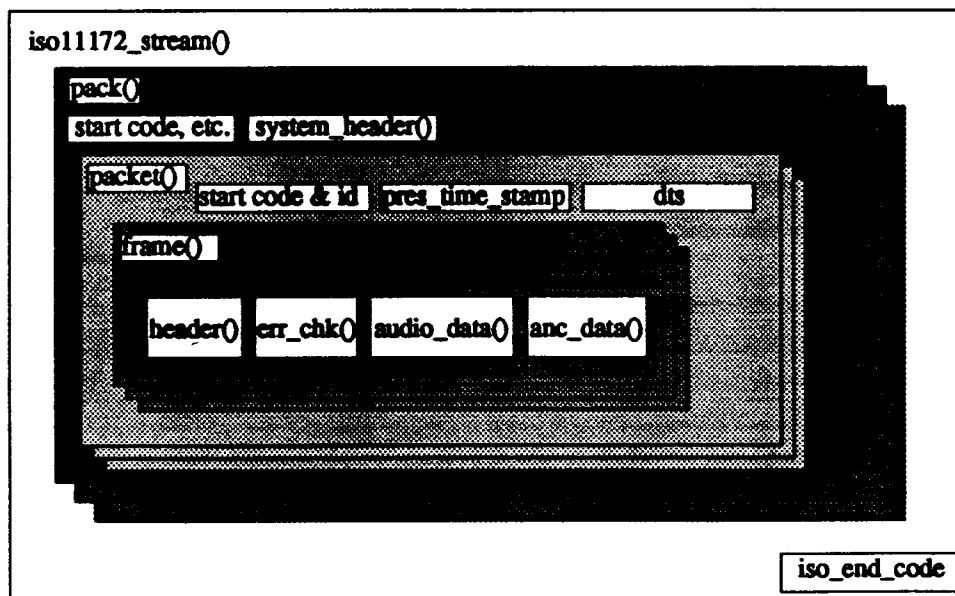
MPEG is also good for video databases for history and education, interactive video games and anyone who wants to add video clips to his application. Commodore is quite excited to be a part of the new digital video revolution.

Three parts: System, Video and Audio.

There are three parts to the MPEG standard: System Layer, Video Layer and Audio Layer. Let's explore them one at a time.

1. System The system layer multiplexes the audio and video layers into a single bit stream. MPEG only defines a bit stream, not the format of the digital storage media. An example of MPEG systems stream is best described in pictures (see figure 1).

Figure 1



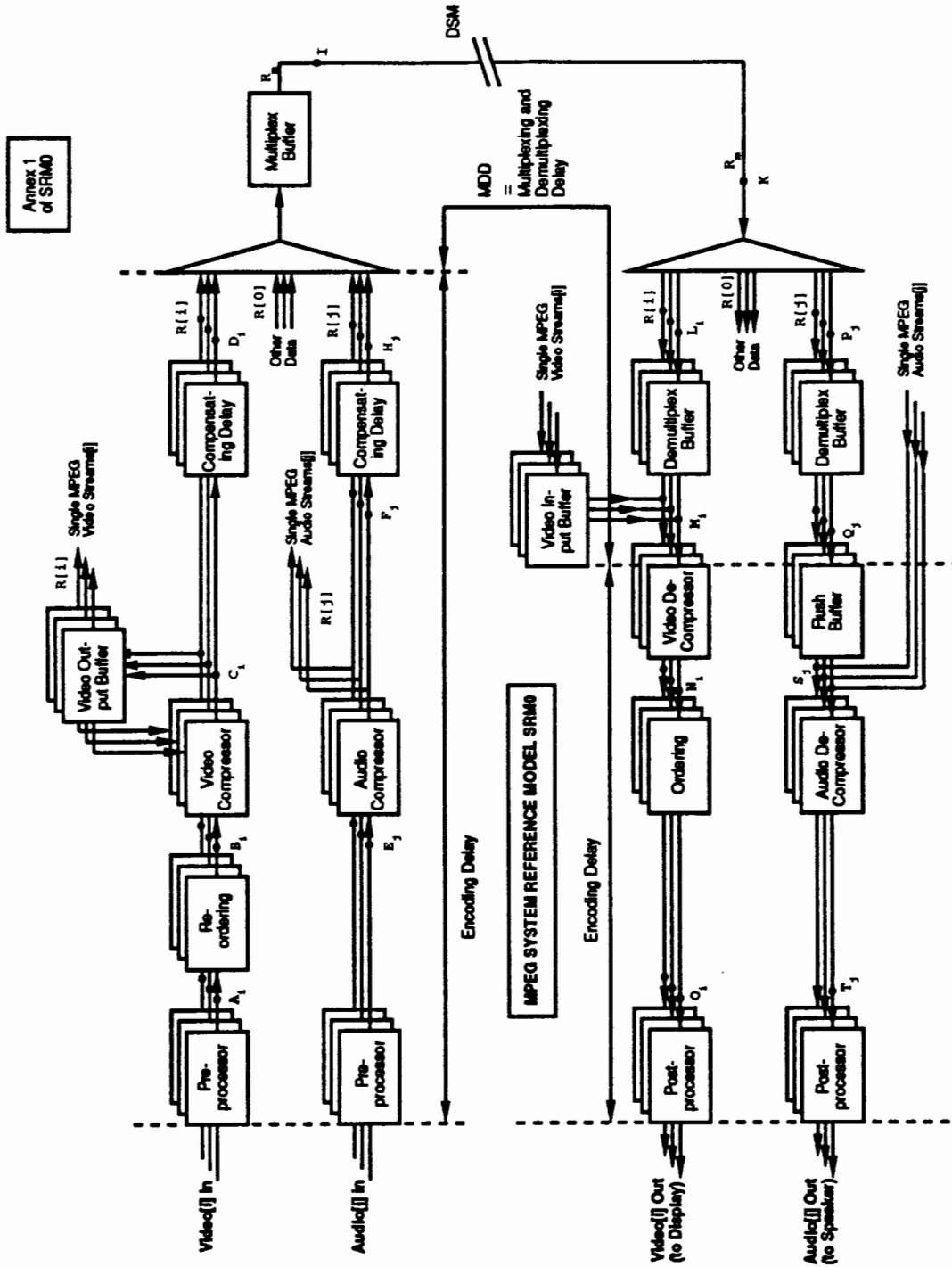
The MPEG system stream is called an `iso11172_stream()` and it consists of a variable number of `pack()`s and an end code. A `pack()` consists of the following sections:

- A. A `pack_start_code` and system clock reference (SCR) information.
- B. A `system_header()` that includes a start/sync code, `packet_length`, stream id and a few misc. items.
- C. A variable number of `packet()`s. Each `packet()` contains:
 - 1. A Packet Header that has a start code, a `stream_id` that identifies if this is a video sequence or audio packet, and optional 32 bit `presentation_time_stamp` (PTS) and `decoding_time_stamp` (DTS).
 - I. For Audio, each Packet Header is followed by a variable number of `frame()`s. An audio frame contains:
 - a. A header/sync word which includes the bit rate and the sampling frequency
 - b. Optional CRC check bits
 - c. The audio data
 - d. Optional user definable ancillary data of definable length
 - II. For video, each Sequence Header is followed by a variable number of groups of pictures. A group of pictures, as the name suggests, consists of one or more individual pictures. The sequence may contain additional sequence headers. A sequence is terminated by a `sequence_end_code`. The Sequence Header specifies the following parameters:
 - a. Bit Rate
 - b. Picture Rate
 - c. Picture Resolution
 - d. Picture Aspect Ratio

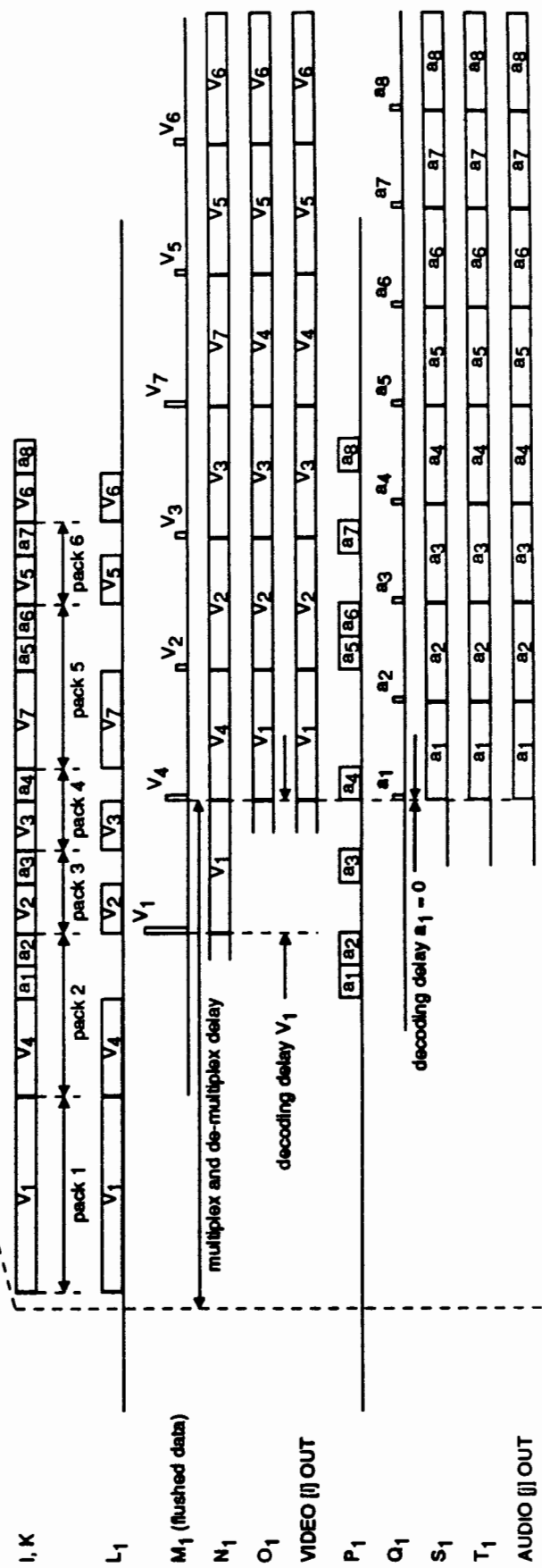
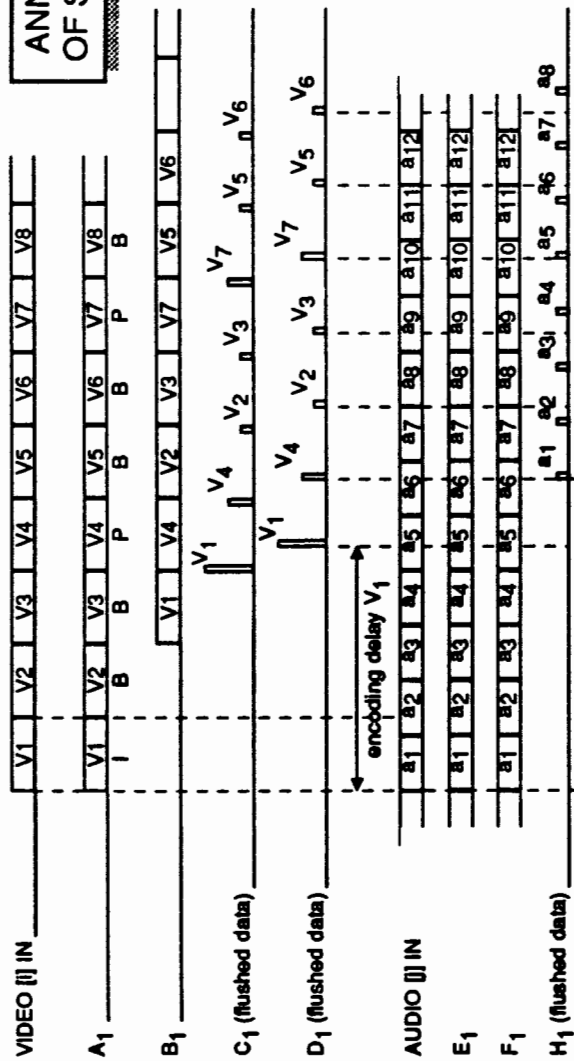
Note: The ancillary data comes with each `frame()`, while the PTS comes with each `packet()`. The CRC Check bits are only for checking the header and do not apply to the data. If certain parameters specified in the bit stream fall within predefined limits, then the bit stream is called a constrained parameter bit stream which should guarantee that all decoders can decode the bit stream.

Another way to show how the system stream is assembled is to show another picture. Figure 2 shows a block diagram of a generalized encoder and decoder. Figure 3 shows what the audio and video data look like at each of the steps along the encoding and decoding path. To fully understand how the audio and video are multiplexed, we need to first understand the Video layer.

Figure 2

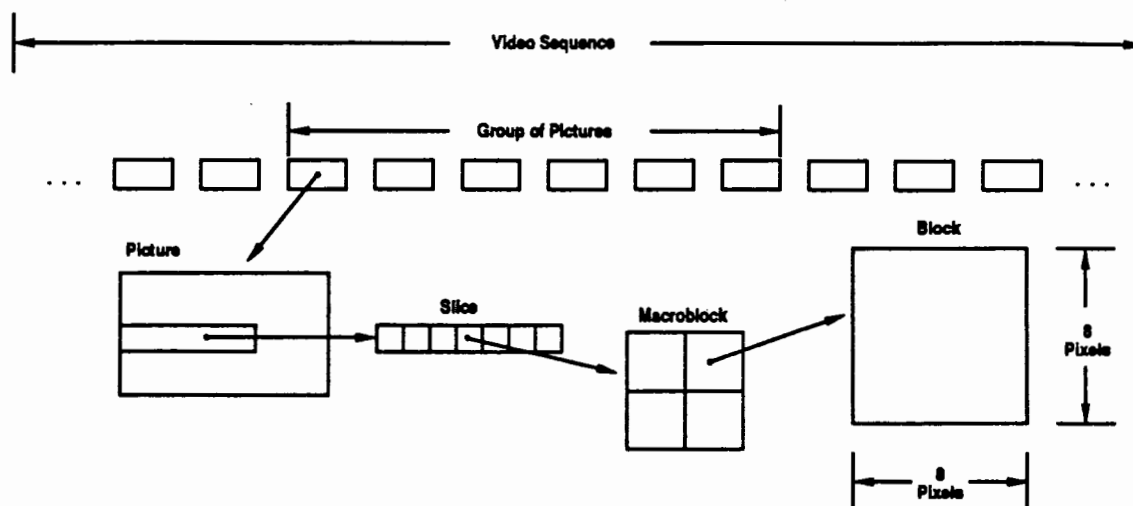


ANNEX 2 OF SRM 0



2.VIDEO The MPEG standard defines a hierarchy of data structures in the video stream as described as follows and as shown in Figure 4.

Figure 4



Video Sequence

Consists of a sequence header, one or more groups of pictures, and an end-of-sequence code. The video sequence is another term for a video stream as defined above.

Group of Pictures

A series of one or more pictures intended to allow random access into the sequence.

Picture

The primary coding unit of a video sequence. A picture consists of three rectangular matrices representing luminance (Y) and two chrominance (CbCr) values. The Y matrix has an even number of rows and columns. The Cb and Cr matrices are one-half the size of the Y matrix in each direction (horizontal and vertical).

Figure 5 shows the relative x-y locations of the luminance and chrominance components. Note that for every four luminance values, there are two associated chrominance values: one Cb value and one Cr value. (The location of the Cb and Cr values is the same, so only one circle is shown in the figure.)

Group of Pictures

A series of one or more pictures intended to allow random access into the sequence.

Picture

The primary coding unit of a video sequence. A picture consists of three rectangular matrices representing luminance (Y) and two chrominance (CbCr) values. The Y matrix has an even number of rows and columns. The Cb and Cr matrices are one-half the size of the Y matrix in each direction (horizontal and vertical).

Figure 5 shows the relative x-y locations of the luminance and chrominance components. Note that for every four luminance values, there are two associated chrominance values: one Cb value and one Cr value. (The location of the Cb and Cr values is the same, so only one circle is shown in the figure.)

Figure 5

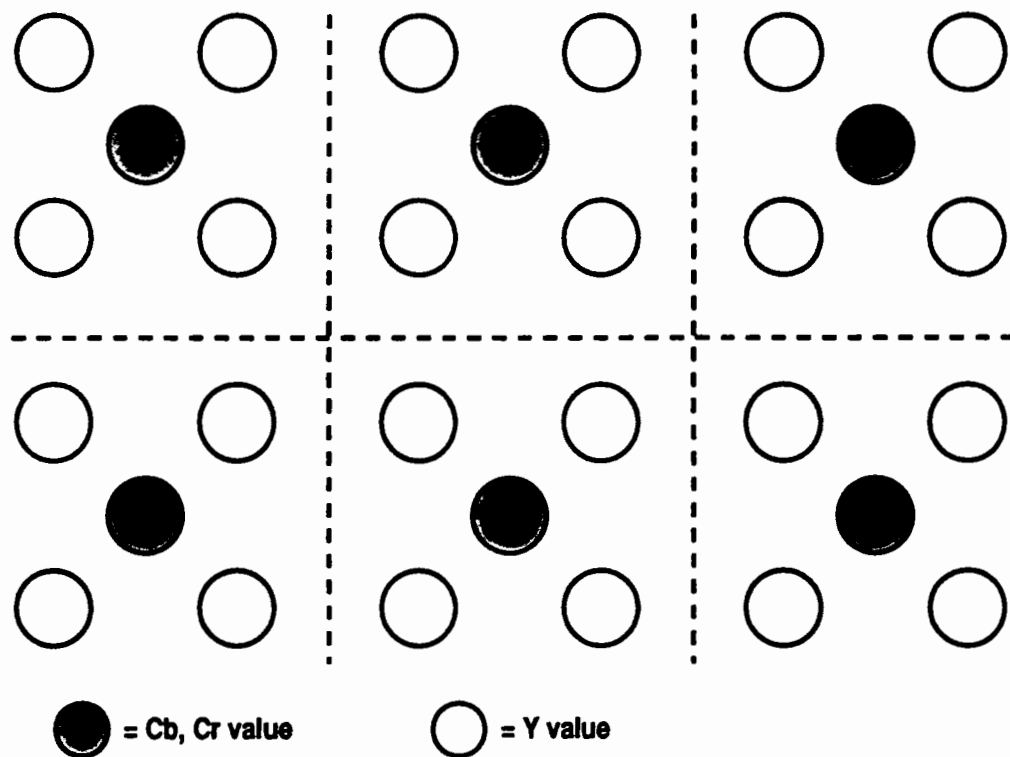
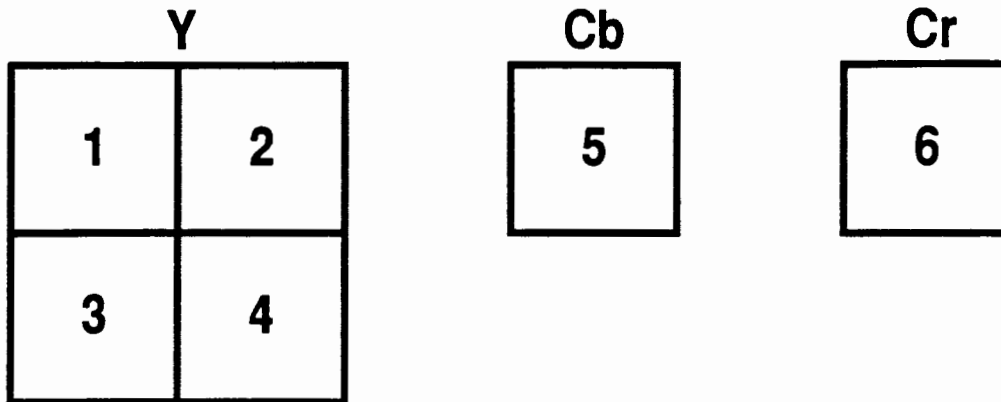


Figure 6



Block

A block is an 8 by 8 set of values of a luminance or chrominance component. Note that a luminance block corresponds to one-fourth as large a portion of the displayed image as does a chrominance block.

Inter-picture Coding

Much of the information in a picture within a video sequence is similar to information in a previous or subsequent picture. The MPEG standard takes advantage of this temporal redundancy to represent some pictures in terms of their differences from reference picture. This section describes the picture types and explains the techniques used in inter-picture coding.

Picture Types

The MPEG standard specifically defines three types of pictures: intra, predicted, and bidirectional.

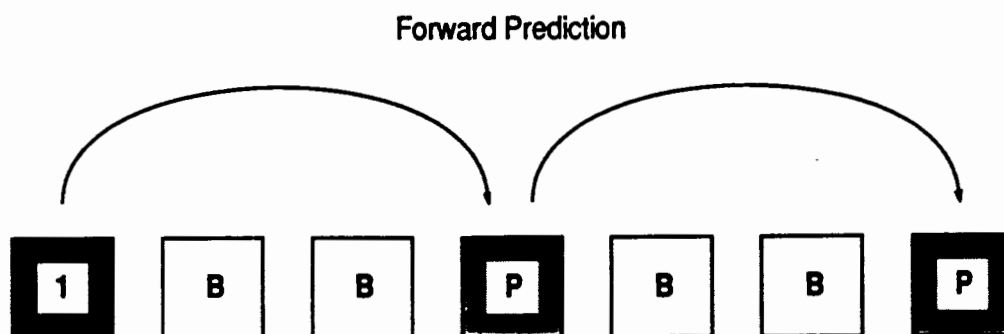
Intra Pictures

Intra or I-pictures are coded using only information present in the picture itself. I-pictures provide random access points into the compressed video data. I-pictures use only transform coding and therefore provide moderate compression. I-pictures typically use about two bits per coded pixel.

Predicted Pictures

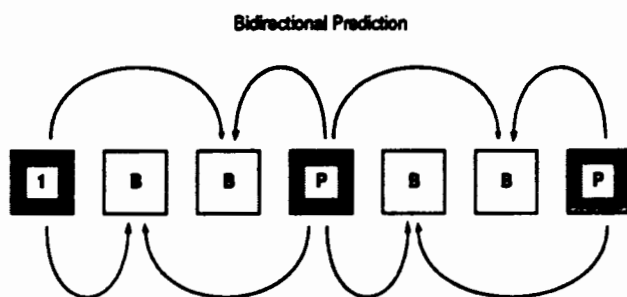
Predicted or P-pictures are coded with respect to the nearest previous I- or P-picture. This technique is called forward prediction and is illustrated in Figure 7. Predicted pictures provide more compression and serve as a reference for B-pictures and future P-pictures. P-pictures use motion compensation to provide more compression than is possible with I-pictures. P-pictures can propagate coding errors, since P-pictures can be predicted from previous P-pictures.

Figure 7



Bidirectional Pictures

Figure 8



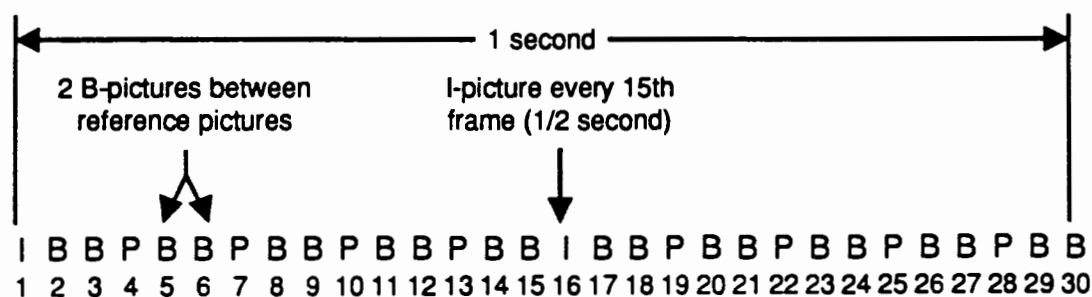
Bidirectional or B-pictures are pictures that use both a past and future picture as a reference. This technique is called bidirectional prediction and is illustrated in Figure 8. Bidirectional pictures provide the most compression and do not propagate errors because they are never used as a reference. Bidirectional prediction also decreases the effect of noise by averaging two pictures.

Video Stream Composition

The MPEG algorithm allows the encoder to choose the frequency and location of I-pictures. This choice is based on the application's need for random accessibility and the location of scene cuts in the video sequence. In applications where random access is important, intra pictures are typically used two times a second.

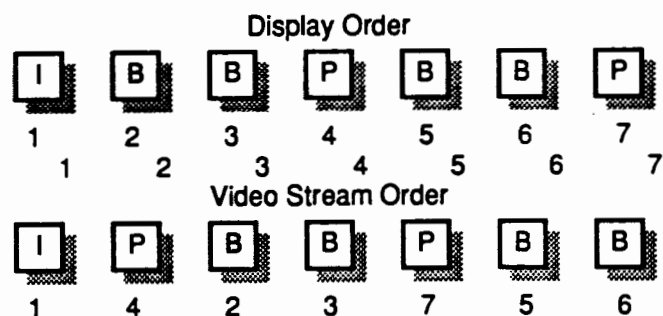
The encoder also chooses the number of bidirectional pictures between any pair of reference (I or P) pictures. This choice is based on factors such as the amount of memory in the encoder and the characteristics of the material being coded. For a large class of scenes, a workable arrangement is to have two bidirectional pictures separating successive reference pictures. A typical arrangement of I-, P-, and B-pictures is shown in Figure 9 in the order in which they are displayed.

Figure 9



The MPEG encoder reorders pictures in the video stream to present the pictures to the decoder in the most efficient sequence. In particular, the reference pictures needed to reconstruct B-pictures are sent before the associated B-pictures. Figure 10 demonstrates this ordering for the first section of the example shown above.

Figure 10



Motion Compensation

Motion compensation is a technique for enhancing the compression of P- and B-pictures by eliminating temporal redundancy. Motion compensation typically improves compression by about a factor of three compared to intra-picture coding. Motion compensation algorithms work at the macroblock level.

When a macroblock is compressed by motion compensation, the compressed file contains this information:

- The spatial difference between the reference and macroblock being coded (motion vectors)
- The content differences between the reference and macroblock being coded (error terms)

Not all information in a picture can be predicted from a previous picture. Consider a scene in which a door opens. The visual details of the room behind the door cannot be predicted from a previous frame in which the door was closed. When a macroblock in a P-picture cannot be represented by motion compensation, it is coded in the same way as a macroblock in an I-picture, that is, by transform coding techniques (see next section on Intra-picture Coding).

Macroblocks in a B-picture can be coded using either a previous or future reference picture as a reference, so that four codings are possible:

- Intra coding: no motion compensation
- Forward prediction: the closest previous I- or P-picture is used as a reference
- Backward prediction: the closest future I- or P-picture is used as a reference
- Bidirectional prediction: two pictures are used as reference, the closest previous I- or P-picture and the closest future I- or P-picture

Backward prediction can be used to predict uncovered areas that do not appear in previous pictures.

Intra-picture (Transform) Coding

The MPEG transform coding algorithm includes these steps:

- Discrete cosine transform (DCT)
- Quantization
- Run-length encoding

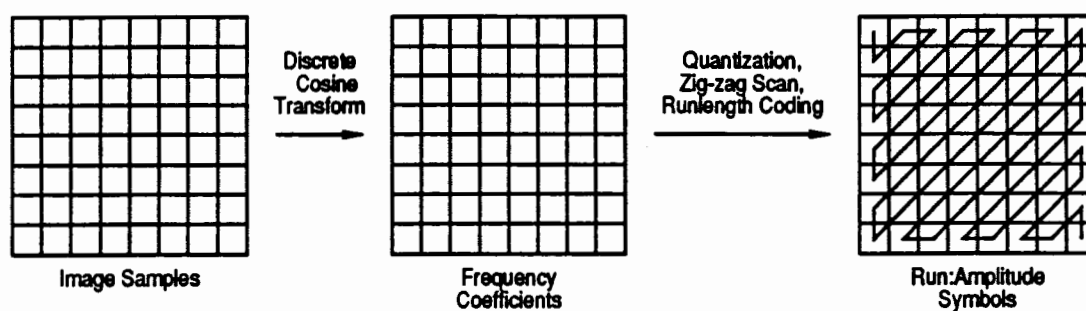
Both image blocks and prediction-error blocks have high spatial redundancy. To reduce this redundancy, the MPEG algorithm transforms 8 x 8 blocks of pixels or 8 x 8 blocks of error terms to the frequency domain with the Discrete Cosine Transform (DCT).

Next, the algorithm quantizes the frequency coefficients. Quantization is the process of approximating each frequency coefficient as one of a limited number of allowed values. The encoder chooses a quantization matrix that determines how each frequency coefficient in the 8 x 8 block is quantized. Human perception of quantization error is lower for high spatial frequencies, so high frequencies are typically quantized more coarsely (i.e., with fewer allowed values) than low frequencies.

The combination of DCT and quantization results in many of the frequency coefficients being zero, especially the coefficients for high spatial frequencies. To take maximum advantage of this, the coefficients are organized in a zigzag order to produce long runs of zeros (see Figure 11). The coefficients are then converted to a series of run-amplitude pairs, each pair indicating a number of zero coefficients and the amplitude of a non-zero coefficient. These run-amplitude pairs are then coded with a variable-length code, which uses shorter codes for commonly occurring pairs and longer codes for less common pairs.

Some blocks of pixels need to be coded more accurately than others. For example, blocks with smooth intensity gradients need accurate coding to avoid visible block boundaries. To deal with this inequality between blocks, the MPEG algorithm allows the amount of quantization to be modified for each 16 x 16 block of pixels. This mechanism can also be used to provide smooth adaptation to a particular bit rate.

Figure 11



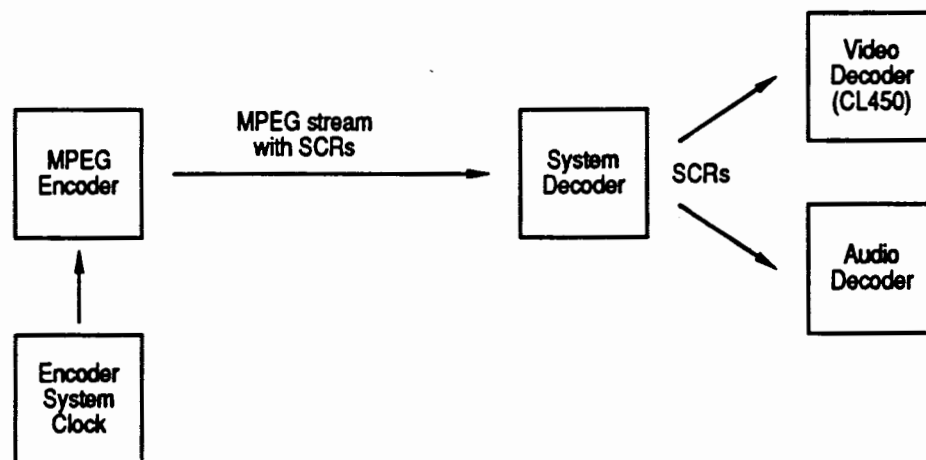
The MPEG standard provides a timing mechanism that ensures synchronization of audio and video. The standard includes two parameters used by the video decoder chip: the system clock reference (SCR) and the presentation time stamp (PTS).

The MPEG system clock running at 90 kHz generates 7.8×10^9 clocks in a 24-hour day. System clock references and presentation time stamps are 33-bit values, which can represent any clock cycle in a 24-hour period.

System Clock References

A system clock reference is a snapshot of the encoder system clock. The SCRs used by the audio and video decoder must have approximately the same value. To keep their values in agreement, SCRs are inserted into the MPEG stream at least as often as every 0.7 seconds by the MPEG encoder, and are extracted by the system decoder and sent to the audio and video decoders as illustrated in Figure 12. The video and audio decoders update their internal clocks using the SCR value sent by the system decoder.

Figure 12



Presentation Time Stamps

Presentation time stamps are samples of the encoder system clock that are associated with some video or audio presentation units. A presentation unit is a decoded video picture or a decoded audio time sequence. The encoder inserts PTSs into the MPEG stream at least as often as every 0.7 seconds. The PTS represents the time at which the video picture is to be displayed or the starting playback time for the audio time sequence.

The video decoder either deletes or repeats pictures to ensure that the PTS matches the current value of the SCR when a picture with a PTS is displayed. If the PTS is earlier (has a smaller value) than the current SCR, the video decoder discards the picture. If the PTS is later (has a larger value) than the current SCR, the video decoder repeats the display of the picture.

3. AUDIO: MPEG has three audio layers. Layers 1 and 2 are most common and are based on a psychoacoustical model of the ear as well as coding and quantization to removed the sound that the human ear can't hear. The bit rates for MPEG audio are between 32Kbps to 192Kbps per channel and the standard support three different Layers. Layer 3 was an attempt to keep the same quality of Layers 1 & 2 at half the bit rate, but many companies have only implemented Layers 1 and 2 due to the complexity of implementing layer 3. Commodore will probably never support Layer 3.

There are four different modes possible for MPEG audio: single channel, dual channel (two independent audio signals coded within one bit stream), stereo (left and right signals of a stereo pair coded within one bit stream), and Joint Stereo (left and right signals of a stereo pair coded within one bit stream with the stereo irrelevancy and redundancy exploited). Depending on the application, different layers of the coding system with increasing encoder complexity and performance can be used.

Layer 1 contains the basic mapping of the digital audio input into 32 subbands, fixed segmentation to format the data into blocks, a psychoacoustical model to determine the adaptive bit allocation, and quantization using block companding and formatting. Layer 2 provides additional code of bit allocation, scale factors, and samples. Layer 3 introduces increased frequency resolution based on a hybrid filter bank. It adds a different (non-uniform) quantizer, adaptive segmentation, and entropy coding of the quantized values. Joint Stereo coding can be added as an additional feature to any of the layers.

In layers 1 and 2, the joint stereo mode is coded as intensity stereo; in layer 3, joint stereo may be coded as either intensity stereo or MS-stereo. Intensity stereo describes a way of keeping the higher frequencies (above about 2KHz) separate, but summing the lower frequencies of the left and right channels to reduce the bit rate. (for instance, most sub woofers on a stereo system are usually monophonic). MS-Stereo is based on coding the sum and difference signal instead of the individual right and left channels (much like a stereo FM radio broadcast is today).

What does this mean for the Amiga?

Since MPEG is an international standard, "software developers of the future" like CNN, the major TV networks, the movie companies and record companies will be digitizing their film

and video libraries in the future. They don't wish to have 6 different versions of the same information, there an international standard is needed. This is what MPEG is attempting to address. Commodore by itself would not be able to convince the major record companies to encode their music videos only for the Amiga, but with MPEG, we will be able to take advantage of this new digital video revolution.

MPEG does not specify a digital storage media. MPEG streams could be on Cable, CD, RF, Ethernet, hard drives, floppies, or anything. For Commodore's purposes, we care mostly about the compact disc. There are generally two types of MPEG CDs: *generic discs* that contain no binary executables for any platform, and *platform specific discs* that contain an executable for a given platform. Generic discs are things like Music Video Discs and Movie Discs. Platform specific discs could be, for example, the Xiphias TimeTable of History discs where John F. Kennedy and Neil Armstrong are shown as MPEG clips, but the disc would also contain an application program (perhaps even multiplatform) for searching and retrieving information on a CDTV or Amiga (or other).

Philips is a member of the MPEG committee, but they are also responsible for licensing companies that make CDs or CD Players. They have published preliminary extensions to the "Green Book" and preliminary specs with JVC for an MPEG Karaoke format. The Green Book discs are obviously platform specific, but the Karaoke Discs are fairly generic. Philips has, however, insisted that all Karaoke Discs (as well as all PhotoCD Discs) have an executable bootable CD-I player program on the disc. As far as other platform vendors (such as Commodore), we can simply ignore that data. The MPEG committee has not addressed a generic CD format at this moment, and though we may be able to adopt one in the future, Philips could get away this.

Commodore's plans

Commodore would like to have full motion video capability across our product line. To cover the different platforms, we can design different form factors for MPEG boards, and in the long term, the MPEG capabilities could be added to the motherboard. We have no specific product announcements, dates or prices at this time.

Hardware Design Points

There are different hardware design points that are needed to cover the spectrum of MPEG encoding and decoding on the Amiga or CDTV.

A low cost decoder card for CDTV and A1200 class systems

This board would contain an MPEG video decoder chip and an MPEG audio decoder chip. The system layer decoding would be handled by the host processor. No scaling would be provided in this system, and it would only support 15KHz video modes. This is the lowest cost hardware configuration for MPEG playback. Because the configuration does not support vertical and horizontal scaling, the images must be encoded in the desired resolution. Most of this time, this is not a great hardship. The MPEG video is genlocked with the Amiga video without the need of an external genlock. Simply use the programmable transparency options of ECS Denise or AA Lisa to have the video show through at the right place.

The MPEG hardware will allow you the ability to move the upper left corner of the MPEG image to anywhere on the screen to simulate movable picture in a picture, but if the window is resized, the image is only clipped, not scaled. The MPEG audio is either digitally muxed with the CD-ROM DAC (which is then added to the Amiga audio) or a separate audio DAC is used and summed with the Amiga's audio.

A higher cost decoder card for the A2000, A3000 and A4000

A higher cost decoder on a Zorro II video card can be designed for desktop Amigas like the A2000, A3000, or A4000. This board would contain the same circuitry as above, with the addition of the AUTOCONFIG logic and extra hardware to perform vertical and horizontal scaling of the image in both 15 and 31KHz. This is a bit tricky and in fact, we are not really sure how to do this yet, but for Amigas with slots, this sort of capability is probably required. What we can show you today is a technology demonstration of a simple MPEG decoder (without scaling) on a ZorroII video card. It is not intended to be a product at this time.

For authoring, a real time digitizer/encoder board is needed for desktop Amigas. This board would allow the user to merely plug in the video from his camcorder or laser disc player and record video directly to hard drive in MPEG format. No one is doing this today, and it may be near the end of 1993 before this capability exists. We are working closely with C-Cube and SGS-Thomson to define this "ideal" solution. This same card should also be capable of decoding, however scaling is not a requirement in an authoring environment (a full size, non-scaled, WYSIWYG system is most likely preferred). For audio authoring, the AT&T DSP3210 card for the Amiga is the preferred platform. Real time MPEG audio encoding and decoding software exist from AT&T under VCOS. Various third party products will likely be adapted to edit the audio prior to encoding. Various third party products will likely be written to edit the video after encoding (probably on a GOP basis). The system layer will be encoded by the host Amiga. For synchronization of audio and video, the audio clock is normally used as the master, and video frames are either dropped or duplicated to adjust synchronization from time to time.

The software support for MPEG on the Amiga also comprises many different parts to support the various hardware configurations described above.

First, an mpeg.library will need to be specified similar to the "Green Book" FMV extensions for CD-I. This library will contain hardware independent routines for controlling the MPEG decoding hardware. These commands include, but are not limited to: Abort, Set Border Color, Change Speed, Close, Conceal, Continue, Create, Freeze, Hide, Image Size, Info, Loop, Next, Off, Set Origin, Pause, Play, Set Position, Release, Select Stream, Show, Status, Trigger, Define Window, and Read or Write Options. In addition, the library must also perform the functions required for decoding the system layer into separate audio and video layers.

In addition to the mpeg.library, device drivers will need to be written for the various hardware devices such as the C-Cube CL450 video decoder chip (e.g., CL450.device) or the LSI-Logic L64111 audio decoder chip (e.g., L64111.device).

For encoding, a set of authoring tools needs to be developed similar to the tools that Carl Sassenrath has developed for CDXL and AVM. A user interface should be created that allows developers to cut and paste audio and video sequences. MPEG encoding parameters such as bit rate for both audio and video, input picture size and frame rate, output compressed picture size and frame rate, picture softness, temporal filtering, interfield filtering, inverse 3:2 pulldown (for dealing with 24 frames per second film), maximum motion vector size, P frame motion vector search range, B frame motion vector search range, buffer size, reference distance (M), Group of Picture Size (N), as well as input and output file names and input devices, will need to be specified at encode time.

At present, only software encoders exist that cannot encode data in real time. Generally they work on high powered Sun Workstations and take many hours per minute of video to encode. To get the best video quality, the source material (such as a one inch video tape or BetaCam tape) is digitized in its full glory on an Abekas, which is only able to store 50 seconds on an 8mm Exabyte tape. These tapes are fed to the Sun one by one, and the video is encoded. We are currently evaluating if a smaller scale software encoder is feasible on the Amiga. For instance, to make a single shot laser disc costs about \$300, then take any of the available Amiga digitizer cards, a laser disc player, and some software to suck in the video frame by frame. If the encoder software were ported to the Amiga, then it would be possible to let a 68040 crunch on the video to encode the sequence. The performance that we would get with this setup is yet to be determined.

In the longer term, the real time hardware encoder card described above will provide the kind of performance that we all desire. Theoretically, the same user interface can be used between the software only encoder and the hardware accelerated encoder. We hope that before the

end of 1993 we will be able to give you this solution. In the mean time, we are investigating the software only solution, and also the possibility of using service bureaus for MPEG encoding at some \$X per minute. The range of X has yet to be determined.

For simple playback of generic MPEG discs, the CDTV player program will need to be modified. Today, we can play Audio CDs and CD+G discs. In the future, the player program should recognize and play generic MPEG movie/music video discs, as well as Karaoke Discs with the Amiga to provide the lyrics in an overlay and PhotoCD Discs.

Conclusion

The market for black boxes that hook to your television set in the living room is getting more and more crowded every day, yet no one has found the holy grail to make the industry take off. MPEG full motion video is the Holy Grail. This is what the average person is expecting of the technology. The Amiga and CDTV are poised to take full advantage of this boom in digital video, from both an authoring point of view and low cost playback. We hope that this introduction to MPEG has been informative and that you start planning your new killer apps that include MPEG full motion video.

Bibliography

1. MPEG1 Standard
The Coding of Moving Pictures and Associated Audio for Digital Storage
Media up to about 1.5 megabits per second
April 1992
Draft International Standard ISO/IEC DIS 11172
ISO/IEC JTC1/SC29/WG11
2. Le Gall, D. J.
MPEG: A video Compression Standard for Multimedia Applications
Communications of the ACM
Volume 34, Number 4, April 1991
3. Le Gall, D. J.
The MPEG video compression Algorithm
Signal Processing : Image Communication
Volume 4, Number 2, April 1992
4. MacInnis, A. G. The MPEG Systems Coding Specification
Signal Processing: Image Communication
Volume 4, Number 2, April 1992
5. C-Cube Microsystems, Inc.
The CL450 MPEG Video Decoder User's Manual
1778 McCarthy Blvd., Milpitas, CA 95035
6. LSI-Logic Corporation The L64111
MPEG Audio Decoder Data Book
1551 McCarthy Blvd, Milpitas, CA 95035
7. SGS-Thomson Microelectronics
The STi3240 MPEG/H.261 Video Decoder Data Book
July 1992
Grenoble, France

8. Jan van der Meer
MPEG System Reference Model Zero
Contribution to MPEG Systems Group,
January 25, 1991
9. Jan van der Meer
The Full Motion System for CD-I
IEEE Transactions on Consumer Electronics
Volume 38, Number 4, November 1992
10. Philips
Compact Disc Interactive Full Functional Specification
September 1990(aka: The Green Book)
11. Philips
Full Motion Extension (tentative)
CD-I Full Functional Specification
July 1992(aka: The Green Book FMV Extensions)
12. Philips & JVC
System Description Karaoke CD (tentative) version 0.9
September 1992
13. Philips & Kodak
System Description PhotoCD (tentative) version 0.9
January 1992





AmigaVision Professional

by Cathy Godfrey

Commodore has recently released AmigaVision Professional (version 2.04). Based on (and compatible with) AmigaVision 1.7, this version of the authoring system includes enhancements to old features and introduces new concepts.

The following is a list (and short description) of some of the important changes made to the AmigaVision authoring system.

Explicit and Computed Referencing

The CALL, CONDITIONAL GOTO and GOTO Icons now support two different types of referencing: Explicit and Computed.

When using Explicit referencing the author selects a specific icon in the flow to reference (this was the normal method of referencing in earlier versions of AmigaVision).

Computed referencing allows the author to specify an icon based on an expression. In the icon's requester, the author enters an expression that evaluates to a string. When AmigaVision executes this Computed Referencing icon, it references the icon whose name equals the string resulting from the expression.

Instead of calling a different subroutine based on a variable (using a row of IFTHEN Icons with associated MODULE and CALL Icons), an author can now use one icon to call any one of several possible choices.

Conditional Interrupts

AmigaVision Professional has a new type of interrupt icon. Like the MOUSE and KEYBOARD INTERRUPT Icons, the CONDITIONAL INTERRUPT Icon contains children that are executed when the interrupt is triggered. Instead of a mouse click or a key press, the Conditional interrupt is triggered when a boolean expression becomes true.

Examples of expressions are:

- ☐ Video() >= 15000
- ☐ Response() == "Help"

Suppose that a kiosk application requires that its attract loop be interrupted every day at 3PM to play a special animation for five minutes. Instead of having to place an icon to check the time at every possible interval, one **CONDITIONAL INTERRUPT** Icon will work. Set the expression in the Conditional interrupt's requester to `clock >= "15:00:00"` and `clock <= "15:05:00"` and everyday at 3:00, the special animation will be played.

Database

When AmigaVision writes or updates database information, the data is temporarily stored in a buffer in RAM. This buffer will be copied to the actual database when the AmigaVision flow quits normally.

This can be a problem for kiosk applications in which the power is simply turned off every night. The AmigaVision flow is not exited normally, so any data in the buffer is lost. Two new features have been added to the R/W Icon to prevent such data loss, **Flush** and **Close Database**.

Flush Database

When AmigaVision Professional executes a R/W Icon set to *Flush Database*, any information stored in the database's RAM buffer is saved to the actual database. This protects it from accidental reboot or power off.

Close Database

If the R/W Icon is set to *Close Database*, AmigaVision saves the database buffer and closes the database. With this command your application can now use more than 10 databases.

Streaming

Normally before an 8SVX or ANIM5 file is played, the entire file must be loaded into memory. AmigaVision Professional supports a new feature called *Streaming*. Streaming saves time and memory by playing the file from its storage location without loading the full file into memory.

In the **SOUND** or **ANIM** Icons a checkmark gadget turns the streaming function on. There is an associated **Buffer** gadget that allows the author to specify the buffer size that will be used during streaming.

The MUSIC Icon

AmigaVision Professional's MUSIC Icon supports playing MIDI files. It can be done in any of three ways: out to a MIDI device, out the Amiga voices, or both MIDI and Amiga.

Note: When you play MIDI out to a MIDI device, be sure "NONE" is specified in the Videodisc Player field of the Video Setup requester.

When playing a MIDI file out the Amiga voices, the author assigns Amiga instrument files to each track of the MIDI file. Not all tracks need be assigned. You can assign only specific tracks to hear only certain parts of the score, like bass or drums.

The MUSIC Icon also allows substituting default instrument for the playback of SMUS files.

CD-DA Playback

The VIDEO Icon has been replaced with the DISC Icon. The DISC Icon not only allows AV Professional to control videodisc players, but it can now control playback of CD digital audio (CD-DA) from CD-ROM.

The CD-DA is controlled by track and index number. The author has control over :

Play

Start the audio.

Pause

Pause audio. A second Pause command will restart the audio from where it was paused.

Stop

Stop the audio.

Volume

Fade the audio to a new volume level. The volume control takes a volume setting and a time value. Volume can be changed immediately, or can be faded-in or faded-out over a period of seconds.

Animation Speed

Normally the speed of an animation is set at the same time it is created. An animation's original speed can now be over-ridden by AV Professional. The number you enter in the Speed gadget of the ANIM Icon actually represents the delay between frames. Increasing the number slows the animation and decreasing the number speeds up the animation.

Computed Quits

The QUIT Icon not only exits entire applications, but it now allows an author to exit a specific section of her flow. To do this, specify an expression in the QUIT Icon which evaluates to a string. When the QUIT Icon is executed, AV will search for the parent icon (MODULE Icon, SCREEN Icon, LOOP Icon, etc.) whose name equals the evaluated expression. If a match is found, the parent icon is exited and the next sibling of that parent icon is executed.

A special feature of this Computed Quit is especially useful when chaining AVf files using the Call Mode. If the module you want to quit happens to be the first MODULE Icon in a called AVf flow, the next sibling of that MODULE Icon is not executed. Instead the flow quits and returns to the calling AVf file. This allows subroutines within an AVf file that is chained using Call mode.

Running Arexx

A Pause gadget on the EXECUTE Icon allows the synchronous or asynchronous execution of ARexx scripts.

Memory Reporting

There are two new features concerning AV Professional's memory usage. The first feature is Memory Usage Reporting. Setting this option will cause AV to open a window after each presentation of a flow. This window will report the maximum amount of memory used to present that flow.

The second feature is a memory limitation requester. The author can specify the amount of Chip and Fast RAM that AmigaVision uses when executing flows.

Object Editor

Control Panel/New Interface

AmigaVision's Object Editor has gone through substantial changes. A Control Panel was added to make selecting editing options faster. The Control Panel contains selection buttons for all objects and all major editing functions. There are X,Y coordinate gadgets that display the selected object's position on the screen. The new Object Editor allows the author to create objects faster and with more accuracy than before.

The following is a list of the new common features available to all objects:

Name Gadget

All objects created in the Object Editor can be given a name. In AV Professional, any object can be address by its name and modified during runtime.

For example, by referring to an object by name, you can modify it's width or height or even remove it from the flow entirely. (See the section on Attributes for more information about modifying objects during runtime.)

X, Y, W, H Gadgets

Each object now displays its own X, Y coordinates. Also available are its height and width. These pieces of information are stored in author-accessible gadgets. If the author needs to align several objects, she can either move the actual object or simply change a number in the X and Y gadgets.

Path

Any object can be assigned a path. Each path point is specified by clicking with the mouse. (Since a path is simply made up of pairs of points, an array can also be used to specify an object's path.)

There are several controls available when using a path:

Path Mode

Path Mode determines what happens to the object when it reaches the end of its path (Loop, Bounce, etc.).

Start

Start indicates at which point along the path the object starts its motion.

Reps

Reps defines the number of times the object loops through its path.

Speed

The Speed gadget controls the speed at which the object moves along its path.

Input Fields

There are two major changes to Input Field objects.

Input Font

The author can now specify any font and size to be used as the input font.

Colors

The author has control over the colors of an Input Field's border, text and background. Alternate colors can be set for the selected state of an Input Field (i.e., when a user is typing in the input field).

ANIM Brushes

A new object added to the Object Editor is the ANIM brush. An ANIM brush object can be made a hit box like other objects. The author has control over the number of repetitions to loop the brush and the speed of the looping.

ANIM Brush objects are very effective when used with the path feature.

Creating Super Objects

Another new concept in AmigaVision Professional is the grouping of objects. A temporary collection of objects is called a *Group*, a permanent one is called a *Super Object*. It is very convenient to collect objects into groups when there are several objects you need to move, copy or delete together.

Attributes

Objects Created In The Object Editor

Objects in the Object Editor are now addressable. Any object can be given a name and using this name, certain attributes of the object can be modified during runtime. For example, some addressable attributes are: screen position, width, height, and colors.

When the same attribute applies to multiple object types, the attribute's name is used consistently (TopEdge and LeftEdge are valid for all display objects, like Circles and Polygons). Some attributes are specific for only one type of object (PageUp and PageDown are only valid for the Text Window object).

Functions are used to manipulate the attributes of objects. For example:

ObjSet()

Used to modify an existing attribute

ObjGet()

Returns the value of an attribute

ObjAdd()

Create an object during runtime

ObjLoad()

Loads a .dob file from a storage device

ObjRemove()

Deletes an object from the application

Below are some examples of expressions which will alter objects. For this example, assume a rectangle object has been created. It sits at screen position 100,100 and the string in its name field is "BigRectangle". A variable called "Result" has been declared of type integer.

ObjSet ("BigRectangle", "TopEdge", 50)

This function call will move the top edge of the object called "BigRectangle" from pixel 100 to pixel 50.

Result = ObjGet ("BigRectangle", "LeftEdge")

The function ObjGet will return the value 100 because the left edge of the object "BigRectangle" is on pixel 100.

ObjClone ("BigRectangle", "NewRectangle")

An exact copy of "BigRectangle" will be made and called "NewRectangle".

ObjRemove("BigRectangle")

The object "BigRectangle" will be deleted from the flow.

The System Object (AVSystem Object)

In addition to the objects that an author creates in the Object Editor, there is one other system object called the AVSystem object (AVSo). This super object contains sub-objects which the author can use to modify the system at a more fundamental level.

The AVSo attributes are addressed using the same functions used for objects created in the Object Editor. The AVSo is initialized by AmigaVision and can be controlled during runtime.

The main AVSo sub-objects are:

Buffering

Controls buffering if IFF files

Pointer

Image, action and position of the AV pointer

Display

Attributes of AV's display screen

Interface

AV's built-in user-interface control

CDTV

CDTV related objects

Some of the most commonly used AVSo attributes are not only addressable through function calls, but have been incorporated into certain icons. For example, the functions of the Buffering sub-object have been duplicated in the ANIM and 8SVX Icons.

Auto-Highlighting Modes

To help AmigaVision-based applications conform to the CDTV user-interface guidelines, two new user-interface modes were added to AmigaVision Professional. These are: automatic highlighting without a visible pointer and automatic highlighting with a pointer.

In the No Pointer mode the AmigaVision pointer is removed from the screen. One hit box is always highlighted (in its Selected state) and movements of the mouse or CDTV remote controller will move the highlight between objects. So, if an object is highlighted (Selected) and the user presses the left arrow button on the remote controller, the original object becomes un-highlighted (returns to its Normal state) and the next available object to the left is highlighted. A press of the mouse button is needed to actually choose the highlighted object.

Pointer mode is very similar to No Pointer mode except that the mouse pointer is not removed from the screen. Objects which contain response strings are highlighted when the pointer moves over them.

Expression Editor

Service Windows

Service windows are used to display information that supplements the normal requester. Unlike normal requesters, they do not prevent operations on their parent requester while they are open. They do not need to be closed individually. When the parent requester closes, all its service windows will close.

The Expression Editor has 5 service windows.

Function

Lists standard functions supported in AV

Variables

Lists user-created variables defined above and to the left of the current icon.

Attributes

Lists all Object attributes.

AVSystem

Lists the AVSystem Super Object and its sub-objects.

Objects

Lists the user-created objects defined above and to the left of the current icon.

Arrays

AV Professional supports multi-dimensional, untyped arrays.

Multi-dimensional means that the author is not limited to one or two dimensional arrays. The number of dimensions used in an AV array is limited only by memory. Untyped means that all the elements in an array doesn't have to be made up of the same type of data. One array element can be of type integer and another of type string.

These arrays are sparsely stored to conserve memory. This means that memory is only allocated for elements as that are used. For example, the expression:

```
TestArray[10, 10, 10] = "Hi, there."
```

results in a three dimensional array with only one element being allocated, instead of a grid of one thousand elements.

Conclusion

As you can see, AmigaVision Professional is more powerful than the earlier versions of AmigaVision in many ways. There are even more features that have not been mentioned here that every author will find useful: New Preferences settings, enhanced printing and searching capabilities, more Command keys in the Flow Editor and Object Editor, etc.





AS225 and SANA II

by Brian Jackson & Greg Miller

AS225 Release 2 - TCP/IP Networking

AS225 was developed in order to give the Amiga connectability with Unix platforms. It is based upon the Berkeley Unix TCP/IP networking code. It uses the Internet Transmission Control Protocol/Internet Protocol (TCP/IP).

History

AS225 came into being as a side benefit of the Amiga Unix project. Release 1 of AS225 provided client services that allowed connection to connect to Unix machines via NFS (Network File System.) There was no NFS file server and many of the included programs were buggy at best. Despite its shortcomings, the software allowed the Amiga to gain admission to places like Stanford Linear Accelerator Center (SLAC), CERN, Moana Kea Observatories, Mount Palomar, Jet Propulsion Laboratories, CalTech (and many other University settings), etc., etc.

What's New?

Release 2 of AS225 adds many new features and updates everything else:

- ☐ An *NFS file server* has been added. This allows the export of volumes on one machine that can be mounted as NFS volumes on other machines. The Amiga can now be used as a native file server.
- ☐ *Domain Name Service* was added to the socket.library so that individual machines need no longer require immense host tables.
- ☐ Commodore is attempting to arrange a deal with SLAC (Stanford Linear Accelerator Center) whereby Willy Langeveld's networking application software can be distributed with Release 2 of AS225.
- ☐ Most of the code has been rewritten to use OS 2.0 system calls wherever appropriate. This means the Release 2 is for systems running at least AmigaOS 2.0.

- ❑ Everything that required it has been rewritten to use the new shared socket library. This made most binaries much smaller (several of the binaries are now under 1K in size.)

- ❑ Additions to the AS225 Release 2 package include:

online

Controls serial interface for SLIP.

offline

Controls serial interface for SLIP.

netrexx

ARexx command host for control of the new NFS file server.

nfsd

NFS file server. This daemon allows you to export volumes on an Amiga to the rest of the world.

Configinet

Controls internal inet.library settings.

sana2 devs

config file for the SANA-II devices (AS225 only.)

tn3270.device (SLAC)

An Amiga device that runs the TCP/IP Telnet protocol. Allows telnet access to any host with a telnetd using a standard, Amiga terminal program.

sendmail & smptd (SLAC)

Mailer and mail server using the standard SMTP protocol.

Lpr (SLAC)

Remote printing from your local spool directory over AS225.

Wprint/WPrintd/WSpooler.rexx (SLAC)

These files make up a simple, configurable network print facility. It allows one to print on a printer attached to the serial or parallel port of one Amiga (the print server) from that Amiga and other Amigas on the net (and also from some UNIX machines). The system has a true spooler, such that there is no conflict if multiple machines try to print at the same time.

What has Changed?

Changes to AS225 from Release 1 include on a file by file basis. This list is neither all inclusive nor definitive since there are several things still in limbo as of this writing :

Rlogin

Rlogin connects your Amiga to a remote host. Changes to rlogin include:

- ☐ 2.0 specific code
- ☐ ReadArgs support.
- ☐ Conclip support.
- ☐ Uses shared socket library
- ☐ Has jump scrolling
- ☐ Menus that allow you to :
 - ☐ turn on/off jump scroll
 - ☐ turn on/off line wrap
 - ☐ set line by line delays when pasting text with conclip.
 - ☐ reset the console device.
 - ☐ reset the display window size to 24 x 80 columns
 - ☐ reset the display width to 80 columns
- ☐ All menu items are controllable via command line options.

Ftp

Ftp is the user interface to the ARPANET standard File Transfer Protocol. It allows the user to transefr files to and from a remote network site. It was completely rewritten for Release 2.

Nfsc

Nfsc is the actual NFS client.

- ☐ Added support for the following 2.0 DOS Packets :
 - ☐ ACTION_FH_FROM_LOCK
 - ☐ ACTION_SAME_LOCK
 - ☐ ACTION_COPY_DIR_FH
 - ☐ ACTION_PARENT_FH
 - ☐ ACTION_EXAMINE_FH
 - ☐ ACTION_SET_FILE_SIZE
- ☐ Raises the maximum number of mounted NFS file systems from 10 to 24.

Nfsmgr

The user front end for nfsc, the NFS client.

- ☐ Updated to use 2.0 functionality.
- ☐ Allows comments in the fstab file.
- ☐ ReadArgs support.
- ☐ Improvements to the command line processing to fix parsing problems.

Chmod

Chmod changes the protection status of a file over NFS.

- ☐ Updated to use 2.0 functionality.
- ☐ Now Allows wildcards.
- ☐ ReadArgs support.

Route

Route is used to manipulate the network routing tables manually.

- ☐ Updated to use 2.0 functionality.
- ☐ ReadArgs support.
- ☐ Shared socket library.
- ☐ ReadArgs support.

Passwd

Passwd is used to change the user's login password.

- ☐ Updated to use 2.0 functionality.
- ☐ Complete rewrite.
- ☐ Shared socket library.
- ☐ ReadArgs support.

Finger

Finger gives you information about the users on the remote site.

- ☐ Updated to use 2.0 functionality.
- ☐ Complete rewrite.
- ☐ Shared socket library.
- ☐ ReadArgs support.

Ping

Ping sends ICMP Echo Request packets to network hosts.

- ☐ Updated to use 2.0 functionality.
- ☐ Complete rewrite.
- ☐ Shared socket library.
- ☐ ReadArgs support.

Showmount

Showmount displays various information about exported and mounted NFS file systems.

- ☐ Updated to use 2.0 functionality.
- ☐ Complete rewrite.
- ☐ Shared socket library.
- ☐ ReadArgs support.

Rsh

Executes a specified command on a remote host.

- ☐ Updated to use 2.0 functionality.
- ☐ Complete rewrite.
- ☐ shared socoket library.
- ☐ ReadArgs support.

Config

Config returuns useful user information and passes user configuration information to the system libraries.

- ☐ Updated to use 2.0 functionality.
- ☐ Complete rewrite.
- ☐ Shared socket library.
- ☐ ReadArgs support.

Boards

Deleted from distribution. Use `sys:Tools/ShowConfig`.

Vers

Deleted from distribution. Use 'version'.

Rshd

The remote shell daemon provides authenticated remote execution facilities.

- ☐ ReadArgs support.
- ☐ Complete rewrite.
- ☐ Uses queue-handler.
- ☐ Updated to use 2.0 functionality.

Inetd

Listener daemon that invokes servers for requested services.

- ☐ complete rewrite.
- ☐ new format for `inet:db/inetd.conf`.
- ☐ new method for launching servers. Programmer docs available.
- ☐ Updated to use 2.0 functionality.

Ftpd

File Transfer Protocol server. Invoked by `inetd`.

- ☐ complete rewrite.
- ☐ Updated to use 2.0 functionality.

Inet.library

- ☐ requires \geq V37 AmigaOS.
- ☐ uses SANA-II devices.
- ☐ Updated to use 2.0 functionality.

Socket.library

- ☐ made a shared library.
- ☐ supports the passing of sockets from one process to another.
(required for `inetd` server launching.)

Developer issues

A networking developer disk will be made available very soon (if it is not available by the time you read this) that will contain the shared socket library, autodocs, includes and examples as well as the same for the Envoy peer-to-peer networking system. As always, CATS is the primary source of information for any such developer materials.

SANA-II: Device Independent Networking

SANA-II provides networking hardware independence for the Amiga. No longer will the end-user have to use hardware type X for his networking needs because it's the only hardware supported by the software he wants to use. Likewise, the end-user will be able to choose from a variety of software options that will all run on his existing hardware. Decisions can be made based upon performance and cost instead of just "what works".

What is SANA-II ?

The goal of SANA-II was to provide networking hardware independence. With SANA-II device drivers for the hardware TCP/IP could run over speaker wire connected to the serial port if have a SANA-II driver for the serial device. ENLan-DFS could run over PCMCIA Ethernet cards, Appletalk could run over a SCSI network, etc. That's SANA-II in a nutshell.

To quote Randell Jesup's DevCon notes for his SANA-II talk at the DevCon '91 in Denver ("Low-level Networking: SANA II") :

"The SANA-II device specification represents a Data Link Level interface standard for Amigas. The intent is :

- ☐ For all Amiga network hardware vendors to create a SANA-II device driver for their hardware.
- ☐ For all protocol stack writers to access the networks via SANA-II device drivers.
- ☐ For any protocol stack to work with any SANA-II device driver."

History

SANA-II is the second incarnation of the "Standard Amiga Network Architecture" that was originally proposed by Dale Luck at the Atlanta DevCon in 1990. The original proposal, known simply as "SANA" or "SANA-I", was a preliminary description of an "ongoing research and development project." (See "SANA: Standard Amiga Network Architecture" by Dale Luck in the Atlanta DevCon notes.)

The original SANA specification actually contained two proposals. The first was the use of the Amiga's library and device models to allow applications to transparently communicate with multiple protocol stacks. While this goal is certainly an admirable one that could theoretically be achieved, the time and resources involved would have been greater than the Amiga Networking Group could afford to devote to the project. So, by the time of the next DevCons (Denver & Milan) the original SANA proposal had given way to SANA-II.

SANA-II is actually a formalized specification for the second part of the original SANA proposal. Using the Amiga's software device model, the spec defines a standard software interface between protocol stacks and networking hardware. The device model allows multiple protocol stacks to talk to any given networking hardware without needing to know anything about the hardware itself. This is the gist of SANA-II. A device driver specific to a given hardware device would use a standardized set of commands that any/all protocol stacks could use to access the hardware. This means that protocol programmers don't need to care about the hardware issues. They just write to the SANA-II spec for their hardware access knowing that their software will operate across any current or future hardware that provides a SANA-II driver.

Why use SANA-II ?

SANA-II offers the Amiga end-user community the ability to pick and choose from multiple software and hardware options without having to worry about which software will work with which hardware. Any networking software that talks to the hardware via a SANA-II device driver can run across any hardware that supports SANA-II with a driver. This encourages the development of both software and hardware.

The use of the standard Amiga device driver interface means that Amiga programmers are already familiar with the concepts, making it easier to write software for the Amiga.

Hardware vendors can feel free to develop new networking hardware with only minimal consideration of software issues (beyond the SANA-II driver itself, of course.)

What has changed since the last DevCon?

- ❑ Packet type specification has been drastically simplified. The original standard called for a generalized "Packet Magic" which all drivers and protocols had to deal with, even though few people should ever have to worry about the problem. It could also have specified that there are 802.3 SANA-II drivers and that there are ethernet drivers and that if you want 802.3 and ethernet (even if on the same wire) from the same machine, use two ethernet boards. This didn't make sense as there is little likelihood of multiple protocols needing to use 802.3 frames nor is there much encouragement for hardware manufactures to provide special 802.3 drivers. The current solution keeps the standard simple and allows highly efficient implementations, but it does make ethernet drivers a little more complex and does make using 802.3 frames harder.

- ❑ The original SANA-II device driver specification called for drivers to have no internal buffers and to get all buffers from protocols in the form of a data structure called a NetBuff. Hence, all protocols were required to use NetBufs. This was highly unsatisfactory since most protocols are implemented from an existing code base which includes its own buffer management scheme. NetBufs are removed from the standard and replaced with function callback.
- ❑ The original standard called for an interface to the ability of some hardware to simultaneously accept packets for several hardware addresses. Such a feature is of dubious usefulness. In order to simplify the standard, station aliases are no longer part of the SANA-II Network Device Driver Specification. If station aliasing does turn out to be a useful feature available on some hardware for the Amiga, the standard can easily be extended to re-introduce station aliasing. Remember that all Exec drivers must check for io_Command values not supported by the driver. Hence, SANA-II commands can be added without requiring that existing drivers be updated.
- ❑ Since the IOSana2Req structure had to be changed anyway, many names in <devices/sana2.h> have been changed to be more consistent with other system names. It is believed that global search and replace should make this a mostly trivial change and that the benefits gained from consistent naming outweigh the inconvenience to those few who have existing SANA-II code.
- ❑ Events are now defined as a bit mask rather than as scalars.

Developer support for SANA-II

The Amiga Networking Group has a SANA-II archive that contains the specifications, autodocs, include files, device drivers for the Commodore networking cards (as well as SLIP for serial device) and example source code showing how to write your own device drivers. The example source code is the source to the CBM SLIP driver. Future networking hardware from Commodore will be provided with a SANA-II device driver. In addition, we are working to ensure the development of drivers for devices such as PCMCIA networking cards and other third party, networking hardware.

Commodore strongly encourages developers of networking software to make use of the SANA-II standard. Commodore also strongly encourages hardware vendors to provide SANA-II device drivers for their hardware. At the time of this writing there are SANA-II drivers for the Commodore A2065 (Ethernet) card, the A2060 (ARCNET) card and the serial

device (both SLIP and CSLIP drivers are available.) Other hardware support is forthcoming both from Commodore and from third-party vendors. The forthcoming AS225 Release 2 and Envoy software packages require SANA-II.

Developers who are writing networking software are encouraged to contact the Amiga Networking Group if they have any questions.

SANA II Autodocs and Include Files

The SANA II autodocs and related include files are listed below.

The autodocs are:

- sana2.device/AbortIO
- sana2.device/CloseDevice
- sana2.device/CMD_CLEAR
- sana2.device/CMD_FLUSH
- sana2.device/CMD_INVALID
- sana2.device/CMD_READ
- sana2.device/CMD_RESET
- sana2.device/CMD_START
- sana2.device/CMD_STOP
- sana2.device/CMD_UPDATE
- sana2.device/CMD_WRITE
- sana2.device/OpenDevice
- sana2.device/S2_ADDMULTICASTADDRESS
- sana2.device/S2_BROADCAST
- sana2.device/S2_CONFIGINTERFACE
- sana2.device/S2_DELMULTICASTADDRESS
- sana2.device/S2_DEVICEQUERY
- sana2.device/S2_GETGLOBALSTATS
- sana2.device/S2_GETSPECIALSTATS
- sana2.device/S2_GETSTATIONADDRESS
- sana2.device/S2_GETTYPESTATS
- sana2.device/S2_MULTICAST
- sana2.device/S2_OFFLINE
- sana2.device/S2_ONEVENT
- sana2.device/S2_ONLINE
- sana2.device/S2_READORPHAN
- sana2.device/S2_TRACKTYPE
- sana2.device/S2_UNTRACKTYPE

The include files are:

- devices/sana2.h
- devices/sana2.i
- devices/sana2specialstats.h
- devices/sana2specialstats.i

SANA II Autodocs

TABLE OF CONTENTS

sana2.device/AbortIO
 sana2.device/CloseDevice
 sana2.device/CMD_CLEAR
 sana2.device/CMD_FLUSH
 sana2.device/CMD_INVALID
 sana2.device/CMD_READ
 sana2.device/CMD_RESET
 sana2.device/CMD_START
 sana2.device/CMD_STOP
 sana2.device/CMD_UPDATE
 sana2.device/CMD_WRITE
 sana2.device/OpenDevice
 sana2.device/S2_ADDMULTICASTADDRESS
 sana2.device/S2_BROADCAST
 sana2.device/S2_CONFIGINTERFACE
 sana2.device/S2_DEMULTICASTADDRESS
 sana2.device/S2_DEVICEQUERY
 sana2.device/S2_GETGLOBALSTATS
 sana2.device/S2_GETSPECIALSTATS
 sana2.device/S2_GETSTATIONADDRESS
 sana2.device/S2_GETTYPESTATS
 sana2.device/S2_MULTICAST
 sana2.device/S2_OFFLINE
 sana2.device/S2_ONEVENT
 sana2.device/S2_ONLINE
 sana2.device/S2_READORPHAN
 sana2.device/S2_TRACKTYPE
 sana2.device/S2_UNTRACKTYPE

sana2.device/AbortIO sana2.device/AbortIO

NAME

AbortIO -- Remove an existing device request.

SYNOPSIS

```
error = AbortIO(Sana2Req)
DO
```

```
LONG AbortIO(struct IOSana2Req *);
```

FUNCTION

This is an exec.library call.

This function aborts an ioRequest. If the request is active, it may or may not be aborted. If the request is queued it is removed. The request will be returned in the same way as if it had normally completed. You must WaitIO() after AbortIO() for the request to return.

INPUTS

Sana2Req - Sana2Req to be aborted.

RESULTS

error - Zero if the request was aborted, non-zero otherwise. io_Error in Sana2Req will be set to IOERR_ABORTED if it was aborted.

NOTES

SEE ALSO

exec.library/AbortIO(), exec.library/WaitIO()

BUGS

```

sana2.device/CloseDevice      sana2.device/CloseDevice
NAME      CloseDevice -- Close the device.
SYNOPSIS
    CloseDevice(Sana2Req)
        A1
    void CloseDevice(struct IOSana2Req *);
FUNCTION
    This function is called by exec.library CloseDevice().
    This function performs whatever cleanup is required at device
    closes.
    Note that all IOREquests MUST be complete before closing. If
    any are pending, your program must AbortIO() then WaitIO() each
    outstanding IOREquest to complete them.
INPUTS
    Sana2Req - Pointer to IOSana2Req initialized by OpenDevice().
NOTES
SEE ALSO
    exec.library/CloseDevice(), exec.library/OpenDevice()
BUGS

```

```

sana2.device/CMD_CLEAR      sana2.device/CMD_CLEAR
NAME      Clear -- Clear internal network interface read buffers.
FUNCTION
    There are no device internal buffers, so CMD_CLEAR does not
    apply to this class of device.
IO REQUEST
    ios2_Command - CMD_CLEAR.
RESULTS
    ios2_Error- IOERR_NOCHMD.
NOTES
SEE ALSO
BUGS

```

<p>sana2.device/CMD_FLUSH</p> <p>NAME Flush -- Clear all queued I/O requests for the SANA-II device.</p> <p>FUNCTION This command aborts all I/O requests in both the read and write request queues of the device. All pending I/O requests are returned with an error message (IOERR_ABORTED). CMD_FLUSH does not affect active requests.</p> <p>IO REQUEST ios2_Command - CMD_FLUSH.</p> <p>RESULTS ios2_Error - Zero if successful; non-zero otherwise.</p> <p>NOTES</p> <p>SEE ALSO</p> <p>BUGS</p>	<p>sana2.device/CMD_INVALID</p> <p>NAME Invalid -- Return with error IOERR_NOCMD.</p> <p>FUNCTION This command causes device driver to reply with an error IOERR_NOCMD as defined in <exec/errors.h> indicating the command is not supported.</p> <p>IO REQUEST ios2_Command - CMD_INVALID.</p> <p>RESULTS ios2_Error - IOERR_NOCMD.</p> <p>NOTES</p> <p>SEE ALSO</p> <p>BUGS Not known to be useful.</p>
---	---

sana2.device/CMD_READ	sana2.device/CMD_READ
NAME	Read -- Get a packet from the network.
FUNCTION	Get the next packet available of the requested packet type. The data copied (via a call to the requester-provided CopyToBuffer function) into los2_Data is normally the Data Link Layer packet data only. If bit SANA2IOB_RAW is set in los2_Flags, then the entire physical frame will be returned. Unlike most Exec devices, SANA-II device drivers do not have internal buffers. If you wish to read data from a SANA-II device you should have multiple CMD_READ requests pending at any given time. The functions provided by you the requestor will be used for any incoming packets of the type you've requested. If no read requests are outstanding for a type which comes in and no read_orphan requests are outstanding, the packet will be lost.
IO REQUEST	los2_Command - CMD_READ los2_Flags - Supported flags are: SANA2IOB_RAW SANA2IOB_QUICK
	los2_PacketType - Packet type desired. los2_Data - Abstract data structure to hold packet data.
RESULTS	los2_Error - Zero if successful; non-zero otherwise. los2_MiscError - More specific error number. los2_Flags - The following flags may be returned: SANA2IOB_RAW SANA2IOB_BCAST SANA2IOB_MCAST
	los2_SrcAddr - Source interface address of packet. los2_DstAddr - Destination interface address of packet. los2_DataLength - Length of packet data. los2_Data - Abstract data structure which packet data is contained in.
NOTES	The driver may not directly examine or modify anything pointed to by los2_Data. It *must* use the requester-provided functions to access this data.
SEE ALSO	S2_READORPHAN, CMD_WRITE, any_protocol/CopyToBuffer
BUGS	

sana2.device/CMD_RESET	sana2.device/CMD_RESET
NAME	Reset -- Reset the network interface to initialized state.
FUNCTION	Currently, SANA-II devices can only be configured once (with CMD_CONFIGINTERFACE) and cannot be re-configured, hence, CMD_RESET does not apply to this class of device.
IO REQUEST	los2_Command - CMD_RESET.
RESULTS	los2_Error - IOERR_NOCMD.
NOTES	
SEE ALSO	
BUGS	

```

sana2.device/CMD_STOP          sana2.device/CMD_STOP
NAME      Stop -- Pause device operation.
FUNCTION
    There is no way for the driver to keep queuing requests without
    servicing them, so CMD_STOP does not apply to this class of
    device. S2_OFFLINE and S2_ONLINE perform a similar function
    to CMD_STOP and CMD_START
IO REQUEST
    los2_Command - CMD_STOP.
RESULTS
    los2_Error- IOERR_NOCHD.
NOTES
SEE ALSO
    S2_ONLINE, S2_OFFLINE
BUGS

```

```

sana2.device/CMD_START          sana2.device/CMD_START
NAME      Start -- Restart device operation.
FUNCTION
    There is no way for the driver to keep queuing requests
    without servicing them, so CMD_STOP does not apply to this
    class of device. S2_OFFLINE and S2_ONLINE perform a
    similar function to CMD_STOP and CMD_START
IO REQUEST
    los2_Command - CMD_START.
RESULTS
    los2_Error- IOERR_NOCHD.
NOTES
SEE ALSO
    S2_ONLINE, S2_OFFLINE
BUGS

```

sana2.device/CMD_UPDATE	sana2.device/CMD_UPDATE
NAME Update -- Force packets out to device.	NAME Write -- Send packet to the network.
FUNCTION Since there are no device internal buffers, CMD_UPDATE does not apply to this class of device.	FUNCTION This command causes the packet to be sent to the specified network interface. Normally, appropriate packet header and trailer information will be added to the packet data when it is sent. If bit SANA2IOB_RAW is set in io_Flags, then the io2_Data is assumed to contain an entire physical frame and will be sent (copied to the wire via CopyFromBuffer()) unmodified.
IO REQUEST io2_Command - CMD_UPDATE.	Note that the device should not check to see if the destination address is on the local hardware. Network protocols should realize that the packet has a local destination long before it gets to a SANA-II driver.
RESULTS io2_Error - IOERR_NOCMD.	IO REQUEST io2_Command - CMD_WRITE. io2_Flags - Supported flags are: SANA2IOB_RAW SANA2IOB_QUICK
NOTES SEE ALSO BUGS	io2_PacketType - Packet type to send. io2_DataAddr - Destination interface address for this packet. io2_DataLength - Length of the Data to be sent. io2_Data - Abstract data structure which packet data is contained in.
	RESULTS io2_Error - Zero if successful; non-zero otherwise. io2_WireError - More specific error number.
	NOTES The driver may not directly examine or modify anything pointed to by io2_Data. It 'must' use the requester-provided functions to access this data.
	SEE ALSO CMD_READ, S2_BROADCAST, S2_MULTICAST, any_protocol/CopyFromBuffer
	BUGS

```

sana2.device/OpenDevice      sana2.device/OpenDevice
NAME      Open -- Request an opening of the network device.
SYNOPSIS  error = OpenDevice(unit, IOSana2Req, flags)
          D0 A1
          BYTE OpenDevice(ULONG, struct IOSana2Req *, ULONG);
FUNCTION  This function is called by exec.library OpenDevice().
          This function performs whatever initialization is required
          per device open and initializes the Sana2Req for use by the
          device.
INPUTS    unit - Device unit to open.
          Sana2Req - Pointer to IOSana2Req structure to be initialized
                  by the sana2.device.
          flags - Supported flags are:
                  SANA2OPB MINE
                  SANA2OPB PROM
          ios2_BufferManagement - A pointer to a tag list containing
                  pointers to buffer management functions.
RESULTS   error - same as io Error
          io_Error - Zero if successful; non-zero otherwise.
          io_Device - A pointer to whatever device will handle the
                  calls for this unit. This pointer may be
                  different depending on what unit is requested.
          ios2_BufferManagement - A pointer to device internal
                  information used to call buffer
                  management functions.
NOTES
SEE ALSO  exec.library/OpenDevice(), exec.library/ClosedDevice()
BUGS

```

```

sana2.device/S2_ADDMULTICASTADDRESS sana2.device/S2_ADDMULTICASTADDRESS
NAME      AddMulticastAddress -- Enable an interface multicast address.
FUNCTION  This command causes the device driver to enable multicast
          packet reception for the requested address.
IO REQUEST
          ios2_Command - S2_ADDMULTICASTADDRESS,
          ios2_SrcAddr - Multicast address to enable.
RESULTS   ios2_Error - Zero if successful; non-zero otherwise.
          ios2_WireError - More specific error number.
NOTES     Multicast addresses are added globally -- anyone using the
          device may receive packets as a result of any multicast address
          which has been added for the device.
          Since multicast addresses are not "bound" to a particular
          packet type, each enabled multicast address has an "enabled"
          count associated with it so that if two protocols add the same
          multicast address and later one removes it, it is still enabled
          until the second removes it.
SEE ALSO  S2_MULTICAST, S2_DELMULTICASTADDRESS
BUGS

```


<p>sana2.device/s2_BROADCAST</p> <p>NAME Broadcast -- Broadcast a packet on network.</p> <p>FUNCTION This command works the same as CMD_WRITE except that it also performs whatever special processing of the packet is required to do a broadcast send. The actual broadcast mechanism is necessarily network/interface/device specific.</p> <p>IO REQUEST</p> <p>ios2_Command - S2_BROADCAST. ios2_Flags - Supported flags are: SANA2IOB_RAW SANA2IOB_QUICK ios2_PacketType - Packet type to send. ios2_DataLength - Length of the Data to be sent. ios2_Data - Abstract data structure which packet data is contained in.</p> <p>RESULTS</p> <p>ios2_DstAddr - The contents of this field are to be considered trash upon return of the IOREQ. ios2_Error - Zero if successful; non-zero otherwise. This command can fail for many reasons and is not supported by all networks and/or network interfaces. ios2_WireError - More specific error number.</p> <p>NOTES</p> <p>The DstAddr field may be trashed by the driver because this function may be implemented by filling DstAddr with a broadcast address and internally calling CMD_WRITE.</p> <p>SEE ALSO</p> <p>CMD_WRITE, S2_MULTICAST</p> <p>BUGS</p>	<p>sana2.device/s2_CONFIGINTERFACE sana2.device/s2_CONFIGINTERFACE</p> <p>NAME ConfigInterface -- Configure the network interface.</p> <p>FUNCTION This command causes the device driver to initialize the interface hardware and to set the network interface address to the address in ios2_SrcAddr. This command can only be executed once and, if successful, will leave the driver and network interface fully operational and the network interface in ios2_SrcAddr.</p> <p>To set the interface address to the factory address, the network management software must use GetStationAddress first and then call ConfigInterface with the result. If there is no factory address then the network software must pick an address to use.</p> <p>Until this command is executed the device will not listen for any packets on the hardware.</p> <p>IO REQUEST</p> <p>ios2_Command - S2_CONFIGINTERFACE. ios2_Flags - Supported flags are: SANA2IOB_QUICK ios2_SrcAddr - Address for this interface.</p> <p>RESULTS</p> <p>ios2_Error - Zero if successful; non-zero otherwise. ios2_WireError - More specific error number. ios2_SrcAddr - Address of this interface as configured.</p> <p>NOTES</p> <p>Some networks have the interfaces choose a currently unused interface address each time the interface is initialized. The caller must check ios2_SrcAddr for the actual interface address after configuring the interface.</p> <p>SEE ALSO</p> <p>S2_GETSTATIONADDRESS</p> <p>BUGS</p>
---	--

sana2.device/s2_DEVICQUERY	sana2.device/s2_DEVICEQUERY
NAME	DeviceQuery -- Return parameters for this network interface.
FUNCTION	This command causes the device driver to report information about the device. Up to SizeAvailable bytes of the information is copied into a buffer pointed to by los2_StatData. The format of the data is as follows:
	<pre> struct Sana2DeviceQuery /* ** Standard information */ ULONG SizeAvailable; /* bytes available */ ULONG SizeSupplied; /* bytes supplied */ LONG DevQueryFormat; /* this is type 0 */ LONG DeviceLevel; /* this document is level 0 */ /* ** Common information */ UNWORD AddrFieldSize; /* address size in bits */ ULONG MTU; /* maximum packet data size */ LONG bps; /* line rate (bits/sec) */ LONG HardwareType; /* what the wire is */ /* ** Format specific information */ }; </pre>
	The SizeAvailable specifies the number of bytes that the caller is prepared to accommodate, including the standard information fields.
	SizeSupplied is the number of bytes actually supplied, including the standard information fields, which will not exceed SizeAvailable.
	<devices/sana2.h> includes constants for these values. If your hardware does not have a number assigned to it, you must contact CATS to get a hardware number.
IO REQUEST	<pre> los2_Command - S2_DEVICEQUERY. los2_StatData - Pointer to Sana2DeviceQuery structure to fill in. </pre>
RESULTS	<pre> los2_Error - Zero if successful; non-zero otherwise. los2_WireError - More specific error number. </pre>
NOTES	
SEE ALSO	
BUGS	

sana2.device/s2_DEMULTICASTADDRESS	sana2.device/s2_DEMULTICASTADDRESS.
NAME	DeMulticastAddress -- Disable an interface multicast address.
FUNCTION	This command causes device driver to disable multicast packet reception for the requested address.
	It is an error to disable a multicast address that is not enabled.
IO REQUEST	<pre> los2_Command - S2_DEMULTICASTADDRESS los2_SrcAddr - Multicast address to disable. </pre>
RESULTS	<pre> los2_Error - Zero if successful; non-zero otherwise. los2_WireError - More specific error number. </pre>
NOTES	<p>Multicast addresses are added globally -- anyone using the device may receive packets as a result of any multicast address which has been added for the device.</p> <p>Since multicast addresses are not "bound" to a particular packet type, each enabled multicast address has an "enabled" count associated with it so that if two protocols add the same multicast address and later one removes it, it is still enabled until the second removes it.</p>
SEE ALSO	S2_ADDMULTICASTADDRESS
BUGS	

```

sana2.device/s2_GETGLOBALSTATS  sana2.device/s2_GETGLOBALSTATS
NAME
    GetGlobalStats -- Get interface accumulated statistics.
FUNCTION
    This command causes the device driver to retrieve various
    global runtime statistics for this network interface. The
    format of the data returned is as follows:
        struct Sana2DeviceStats {
            ULONG PacketsReceived;
            ULONG PacketsSent;
            ULONG BadData;
            ULONG Overruns;
            ULONG UnknownTypesReceived;
            ULONG Reconfigurations;
            timeval lastStart;
        };
IO REQUEST
    ios2 Command - S2_GETGLOBALSTATS.
    ios2_StatData - Pointer to Sana2DeviceStats structure to fill.
RESULTS
    ios2_Error - Zero if successful; non-zero otherwise.
    ios2_WireError - More specific error number.
NOTES
    SEE ALSO
        S2_GETSPECIALSTATS
BUGS

```

```

sana2.device/s2_GETSPECIALSTATS  sana2.device/s2_GETSPECIALSTATS
NAME
    GetSpecialStats -- Get network type specific statistics.
FUNCTION
    This function returns statistics which are specific to the type
    of network medium this driver controls. For example, this
    command could return statistics common to all Ethernets which
    are not common to all network mediums in general.
    The supplied Sana2SpecialStatData structure is given below:
        struct Sana2SpecialStatData {
            ULONG RecordCountMax;
            ULONG RecordCountSupplied;
            struct Sana2StatRecord[RecordCountMax];
        };
    The format of the data returned is:
        struct Sana2StatRecord {
            ULONG Type; /* commodore registered */
            LONG Count; /* the stat itself */
            char *String; /* null terminated */
        };
    The RecordCountMax field specifies the number of records that
    the caller is prepared to accommodate.
    RecordCountSupplied is the number of record actually supplied
    which will not exceed RecordCountMax.
IO REQUEST
    ios2 Command - S2_GETSPECIALSTATS.
    ios2_StatData - Pointer to a Sana2SpecialStatData structure to
        fill.
    RecordCountMax must be initialized.
RESULTS
    ios2_Error - Zero if successful; non-zero otherwise.
    ios2_WireError - More specific error number.
NOTES
    Commodore will maintain registered statistic Types.
    SEE ALSO
        S2_GETGLOBALSTATS, <devices/sana2specialstats.h>
BUGS

```

```

sana2.device/s2_GETSTATIONADDRESS  sana2.device/s2_GETSTATIONADDRESS
NAME  GetStationAddress -- Get default and interface address.
FUNCTION
    This command causes the device driver to copy the current
    interface address into los2_SrcAddr, and to copy the factory
    default station address (if any) into los2_DataAddr.
IO REQUEST
    los2_Command - S2_GETSTATIONADDRESS.
RESULTS
    los2_Error - Zero if successful; non-zero otherwise.
    los2_WireError - More specific error number.
    los2_SrcAddr - Current interface address.
    los2_DataAddr - Default interface address (if any).
NOTES
SEE ALSO
    S2_CONFIGINTERFACE
BUGS

```

```

sana2.device/s2_GETTYPESTATS
sana2.device/s2_GETTYPESTATS
NAME  GetTypeStats -- Get accumulated type specific statistics.
FUNCTION
    This command causes the device driver to retrieve various
    packet type specific runtime statistics for this network
    interface. The format of the data returned is as follows:
    struct Sana2TypeStatData {
        LONG PacketsSent;
        LONG PacketsReceived;
        LONG BytesSent;
        LONG BytesReceived;
        LONG PacketsDropped;
    };
IO REQUEST
    los2_Command - S2_GETTYPESTATS.
    los2_PacketType - Packet type of interest.
    los2_StatData - Pointer to TypeStatData structure to fill in.
RESULTS
    los2_Error - Zero if successful; non-zero otherwise.
    los2_WireError - More specific error number.
NOTES
    Statistics for a particular packet type are only available
    while that packet type is being "tracked".
SEE ALSO
    S2_TRACKTYPE, S2_UNTRACKTYPE
BUGS

```

<p>sana2.device/S2_MULTICAST</p> <p>NAME Multicast -- Multicast a packet on network.</p> <p>FUNCTION This command works the same as CMD_WRITE except that it also performs whatever special processing of the packet is required to do a multicast send. The actual multicast mechanism is necessarily network/interface/device specific.</p> <p>IO REQUEST</p> <p>RESULTS</p> <p>los2_Error - Zero if successful; non-zero otherwise.</p> <p>los2_WireError - More specific error number.</p> <p>NOTES</p> <p>S2ERR_OUTOFSERVICE.</p> <p>While the interface is offline, all read, writes and any other command that touches interface hardware will be rejected with los2_Error set to S2ERR_OUTOFSERVICE.</p> <p>This command is intended to permit a network interface to be tested on an otherwise live system.</p> <p>SEE ALSO</p> <p>S2_ONLINE</p> <p>BUGS</p>	<p>sana2.device/S2_OFFLINE</p> <p>NAME Offline -- Remove interface from service.</p> <p>FUNCTION This command removes a network interface from service.</p> <p>IO REQUEST</p> <p>los2_Command - S2_OFFLINE.</p> <p>RESULTS</p> <p>los2_Error - Zero if successful; non-zero otherwise.</p> <p>los2_WireError - More specific error number.</p> <p>NOTES</p> <p>Aborts all pending reads and writes with los2_Error set to S2ERR_OUTOFSERVICE.</p> <p>While the interface is offline, all read, writes and any other command that touches interface hardware will be rejected with los2_Error set to S2ERR_OUTOFSERVICE.</p> <p>This command is intended to permit a network interface to be tested on an otherwise live system.</p> <p>SEE ALSO</p> <p>S2_ONLINE</p> <p>BUGS</p>
---	--

```

sana2.device/s2_ONEVENT      sana2.device/s2_ONEVENT
NAME      OnEvent -- Return when specified event occurs.
FUNCTION  This command returns when a particular event condition has
          occurred on the network or this network interface.
IO REQUEST
  los2_Command - S2_ONEVENT.
  los2_Flags - Supported flags are:
    SANA2IOB_QUICK
  los2_WireError - Mask of event(s) to wait for (from
    <devices/sana2.h>).
RESULTS
  los2_Error - Zero if successful; non-zero otherwise.
  los2_WireError - Mask of events that occurred.
NOTES
  If this device driver does not understand the specified event
  condition(s) then the command returns immediately with
  los2_Req.io_Error set to S2_ERR_NOT_SUPPORTED and
  los2_WireError S2WERR_BAD_EVENT. A successful return will
  have los2_Error set to zero los2_WireError set to the event
  number.
  All pending requests for a particular event will be returned
  when that event occurs.
  All event types that cover a particular condition are
  returned when that condition occurs. For instance, if an
  error is returned by a buffer management function during
  receive processing, events of types S2EVENT_ERROR, S2EVENT_RX
  and S2EVENT_BUFF would be returned if pending.
  Types ONLINE and OFFLINE return immediately if the device is
  already in the state to be waited for.
SEE ALSO
BUGS

```

```

sana2.device/s2_ONLINE      sana2.device/s2_ONLINE
NAME      Online -- Put a network interface back in service.
FUNCTION  This command places an offline network interface back into
          service.
IO REQUEST
  los2_Command - S2_ONLINE.
RESULTS
  los2_Error - Zero if successful; non-zero otherwise.
  los2_WireError - More specific error number.
NOTES
  This command is responsible for putting the network interface
  hardware back into a known state (as close as possible to the
  state before S2_OFFLINE) and resets the unit global and special
  statistics.
SEE ALSO
  S2_OFFLINE
BUGS

```

<p>sana2.device/S2_READORPHAN sana2.device/S2_READORPHAN</p> <p>NAME ReadOrphan -- Get a packet for which there is no reader.</p> <p>FUNCTION</p> <p>Get the next packet available that does not satisfy any then-pending CMD_READ requests. The data returned in the ios2_Data structure is normally the Data Link Layer packet type field and the packet data. If bit SANA2IOB_RAW is set in ios2_Flags, then the entire Data Link Layer packet, including both header and trailer information, will be returned.</p> <p>IO REQUEST</p> <p>ios2_Command - CMD_READORPHAN. ios2_Flags - Supported flags are: SANA2IOB_RAW SANA2IOB_QUICK ios2_DataLength - Length of the Data to be sent. ios2_Data - Abstract data structure which packet data is contained in.</p> <p>RESULTS</p> <p>ios2_Error - Zero if successful; non-zero otherwise. ios2_WireError - More specific error number. ios2_Flags - The following flags may be returned: SANA2IOB_RAW SANA2IOB_BCAST SANA2IOB_MCAST ios2_SrcAddr - Source interface address of packet. ios2_DstAddr - Destination interface address of packet. ios2_DataLength - Length of the Data to be sent. ios2_Data - Abstract data structure which packet data is contained in.</p> <p>NOTES</p> <p>This is intended for debugging and management tools. Protocols should not use this.</p> <p>As with 802.3 packets on an ethernet, to determine which protocol family the returned packet belongs to you may have to specify SANA2IOB_RAW to get the entire data link layer wrapper (which is where the protocol type may be kept). Notice this necessarily means that this cannot be done in a network interface independent fashion. The driver will, however, fill in the PacketType field to the best of its ability.</p> <p>SEE ALSO CMD_READ, CMD_WRITE</p> <p>BUGS</p>	<p>sana2.device/S2_TRACKTYPE sana2.device/S2_TRACKTYPE</p> <p>NAME TrackType -- Accumulate statistics about a packet type.</p> <p>FUNCTION</p> <p>This command causes the device driver to accumulate statistics about a particular packet type. Packet type statistics, for the particular packet type, are zeroed by this command.</p> <p>IO REQUEST</p> <p>ios2_Command - S2_TRACKTYPE. ios2_PacketType - Packet type of interest.</p> <p>RESULTS</p> <p>ios2_Error - Zero if successful; non-zero otherwise. ios2_WireError - More specific error number.</p> <p>NOTES</p> <p>SEE ALSO S2_UNTRACKTYPE, S2_GETTYPESTATS</p> <p>BUGS</p>
--	---

```

sana2.device/s2_UNTRACKTYPE      sana2.device/s2_UNTRACKTYPE
NAME
    UntrackType -- End statistics about a packet type.
FUNCTION
    This command causes the device driver to stop accumulating
    statistics about a particular packet type.
IO REQUEST
    ios2_Command - S2_UNTRACKTYPE.
    ios2_PacketType - Packet type of interest.
RESULTS
    ios2_Error - Zero if successful; non-zero otherwise.
    ios2_WireError - More specific error number.
NOTES
SEE ALSO
    S2_TRACKTYPE, S2_GETTYPESTATS
BUGS
#define SANA2_SANA2DEVICE_H
#define SANA2_SANA2DEVICE_H 1
/*
** $Filename: devices/sana2.h $
** $Revision: 1.11 $
** $Date: 92/11/10 13:35:29 $
**
** Structure definitions for SANA-II devices.
** (C) Copyright 1991 Commodore-Amiga Inc.
** All Rights Reserved
*/
#ifndef EXEC_TYPES_H
#include <exec/types.h>
#endif
#ifndef EXEC_PORTS_H
#include <exec/ports.h>
#endif
#ifndef EXEC_IO_H
#include <exec/io.h>
#endif
#ifndef EXEC_ERRORS_H
#include <exec/errors.h>
#endif
#ifndef DEVICES_TIMER_H
#include <devices/timer.h>
#endif
#ifndef UTILITY_TAGITEM_H
#include <utility/tagitem.h>
#endif
#define SANA2_MAX_ADDR_BITS (128)
#define SANA2_MAX_ADDR_BYTES ((SANA2_MAX_ADDR_BITS+7)/8)

```

```

struct IOSana2Req {
    struct IORequest ios2_Req; /* wire type specific error */
    ULONG ios2_WireError; /* packet type */
    ULONG ios2_PacketType; /* source addr */
    USHORT ios2_SrcAddr[SANA2_MAX_ADDR_BYTES]; /* dest address */
    USHORT ios2_DstAddr[SANA2_MAX_ADDR_BYTES]; /* length of packet data */
    VOID *ios2_DataLength; /* packet data */
    VOID *ios2_Data; /* statistics data pointer */
    VOID *ios2_StatData; /* see SANA-II OpenDevice adoc */
};

/*
** defines for the io_Flags field
*/
#define SANA2IOB_RAW (7) /* raw packet IO requested */
#define SANA2IOF_RAW (1<<SANA2IOB_RAW)
#define SANA2IOB_BCAST (6) /* broadcast packet (received) */
#define SANA2IOF_BCAST (1<<SANA2IOB_BCAST)
#define SANA2IOB_MCAST (5) /* multicast packet (received) */
#define SANA2IOF_MCAST (1<<SANA2IOB_MCAST)
#define SANA2IOB_QUICK (IOB_QUICK) /* quick IO requested (0) */
#define SANA2IOF_QUICK (IOF_QUICK)

/*
** defines for OpenDevice() flags
*/
#define SANA2OPB_MINE (0) /* exclusive access requested */
#define SANA2OPF_MINE (1<<SANA2OPB_MINE)
#define SANA2OPB_PROM (1) /* promiscuous mode requested */
#define SANA2OPF_PROM (1<<SANA2OPB_PROM)

/*
** defines for OpenDevice() tags
*/
#define S2_Dummy (TAG_USER + 0x80000)
#define S2_CopyToBuff (S2_Dummy + 1)
#define S2_CopyFromBuff (S2_Dummy + 2)

struct Sana2DeviceQuery {
    /* Standard information
    */
    ULONG SizeAvailable; /* bytes available
    */
    ULONG SizeSupplied; /* bytes supplied
    */
    ULONG DevQueryFormat; /* this is type 0
    */
    ULONG DeviceLevel; /* this document is level 0
    */
    /* Common information
    */
    ULONG AddrFieldSize; /* address size in bits
    */
    ULONG MTU; /* maximum packet data size
    */
    ULONG BPS; /* line rate (bits/sec)
    */
    ULONG HardwareType; /* what the wire is
    */
    /* Format specific information
    */
};
/*

```



```

** defined hardware types
** If your hardware type isn't listed below contact CATS to get a
** new type number added for your hardware.
*/
#define S2WireType_Ethernet 1
#define S2WireType_IEEE802 6
#define S2WireType_Arcnet 7
#define S2WireType_LocalTalk 11
#define S2WireType_DyLAN 12
#define S2WireType_AmokrNet 200
#define S2WireType_PPP 253
#define S2WireType_SLIP 254
#define S2WireType_CSLIP 255

struct Sana2PacketTypeStats {
    ULONG PacketsSent; /* transmitted count */
    ULONG PacketsReceived; /* received count */
    ULONG BytesSent; /* bytes transmitted count */
    ULONG BytesReceived; /* bytes received count */
    ULONG PacketsDropped; /* packets dropped count */
};

struct Sana2SpecialStatRecord {
    ULONG Type; /* statistic identifier */
    ULONG Count; /* the statistic */
    char *String; /* statistic name */
};

struct Sana2SpecialStatHeader {
    ULONG RecordCountMax; /* room available */
    ULONG RecordCountSupplied; /* number supplied */
    /* struct Sana2SpecialStatRecord[RecordCountMax]; */
};

struct Sana2DeviceStats {
    ULONG PacketsReceived; /* received count */
    ULONG PacketsSent; /* transmitted count */
    ULONG BadData; /* bad packets received */
    ULONG Overruns; /* hardware miss count */
    ULONG Unused; /* Unused field */
    ULONG UnknownTypesReceived; /* orphan count */
    ULONG Reconfigurations; /* network reconfigurations */
    struct timeval LastStart; /* time of last online */
};

/* Device Commands
*/
#define S2_START (CMD_NONSTD)
#define S2_DEVICEQUERY (S2_START+0)
#define S2_GETSTATIONADDRESS (S2_START+1)
#define S2_CONFIGINTERFACE (S2_START+2)
#define S2_ADDMULTICASTADDRESS (S2_START+5)
#define S2_DELMULTICASTADDRESS (S2_START+6)
#define S2_MULTICAST (S2_START+7)
#define S2_BROADCAST (S2_START+8)
#define S2_TRACETYPE (S2_START+9)
#define S2_UNTRACETYPE (S2_START+10)
#define S2_GETTYPESTATS (S2_START+11)
#define S2_GETSPECIALSTATS (S2_START+12)
#define S2_GETGLOBALSTATS (S2_START+13)
#define S2_ONEVENT (S2_START+14)
#define S2_READORPHAN (S2_START+15)

```

```

#define S2_ONLINE (S2_START+16)
#define S2_OFFLINE (S2_START+17)
#define S2_END (S2_START+18)

/* defined errors for Io_Error (see also <exec/errors.h>)
*/
#define S2ERR_NO_ERROR 0 /* peachy-keen */
#define S2ERR_NO_RESOURCES 1 /* resource allocation failure */
#define S2ERR_BAD_ARGUMENT 3 /* garbage somewhere */
#define S2ERR_BAD_STATE 4 /* inappropriate state */
#define S2ERR_BAD_ADDRESS 5 /* who? */
#define S2ERR_MTU_EXCEEDED 6 /* too much to chew */
#define S2ERR_NOT_SUPPORTED 8 /* hardware can't support cmd */
#define S2ERR_SOFTWARE 9 /* software error detected */
#define S2ERR_OUTOFSERVICE 10 /* driver is OFFLINE */

/* From <exec/errors.h>
*/
#define IOERR_OPENFAIL (-1) /* device/unit failed to open */
#define IOERR_ABORTED (-2) /* request terminated early (after AbortIO()) */
#define IOERR_NOCHD (-3) /* command not supported by device */

#define IOERR_BADLENGTH (-4) /* not a valid length (usually IO LENGTH) */
#define IOERR_BADADDRESS (-5) /* invalid address (misaligned or bad range) */
#define IOERR_UNITBUSY (-6) /* device opens ok, but requested unit is busy */
#define IOERR_SELFTEST (-7) /* hardware failed self-test */

/* defined errors for Io2_WireError
*/
#define S2MERR_GENERIC_ERROR 0 /* no specific info available */
#define S2MERR_NOT_CONFIGURED 1 /* unit not configured */
#define S2MERR_UNIT_ONLINE 2 /* unit is currently online */
#define S2MERR_UNIT_OFFLINE 3 /* unit is currently offline */
#define S2MERR_ALREADY_TRACKED 4 /* protocol already tracked */
#define S2MERR_NOT_TRACKED 5 /* protocol not tracked */
#define S2MERR_BUFF_ERROR 6 /* buff mgt func returned error */
#define S2MERR_SRC_ADDRESS 7 /* source address problem */
#define S2MERR_DST_ADDRESS 8 /* destination address problem */
#define S2MERR_BAD_BROADCAST 9 /* broadcast address problem */
#define S2MERR_BAD_MULTICAST 10 /* multicast address problem */
#define S2MERR_MULTICAST_FULL 11 /* multicast address list full */
#define S2MERR_BAD_EVENT 12 /* unsupported event class */
#define S2MERR_BAD_STATDATA 13 /* statdata failed sanity check */
#define S2MERR_IS_CONFIGURED 15 /* attempt to config twice */
#define S2MERR_NULL_POINTER 16 /* null pointer detected */

/* defined events
*/
#define S2EVENT_ERROR (1L<0) /* error catch all */
#define S2EVENT_TX (1L<1) /* transmitter error catch all */
#define S2EVENT_RX (1L<2) /* receiver error catch all */
#define S2EVENT_ONLINE (1L<3) /* unit is in service */
#define S2EVENT_OFFLINE (1L<4) /* unit is not in service */
#define S2EVENT_BUFF (1L<5) /* buff mgt function error */
#define S2EVENT_HARDWARE (1L<6) /* hardware error catch all */
#define S2EVENT_SOFTWARE (1L<7) /* software error catch all */

#endif /* SANA2_DEVICE_H */

```

```

IFND SANA2 SANA2DEVICE_I 1
SANA2_SANA2DEVICE_I SET
** $Filename: devices/sana2.1 $
** $Revision: 1.12 $
** $Date: 92/11/10 13:36:28 $
**
** Structure definitions for SANA-II devices.
** (C) Copyright 1991 Commodore-Amiga Inc.
** All Rights Reserved
**
IFND EXEC_TYPES_I
INCLUDE "exec/types.1"
ENDC

IFND EXEC_PORTS_I
INCLUDE "exec/ports.1"
ENDC

IFND EXEC_IO_I
INCLUDE "exec/io.1"
ENDC

IFND EXEC_ERRORS_I
INCLUDE "exec/errors.1"
ENDC

IFND DEVICES_TIMER_I
INCLUDE "devices/timer.1"
ENDC

IFND UTILITY_TAGITEM_I
INCLUDE "utility/tagitem.1"
ENDC

SANA2_MAX_ADDR_BITS EQU 128
SANA2_MAX_ADDR_BYTES EQU ((SANA2_MAX_ADDR_BITS+7)/8)

STRUCTURE IOSana2Req,0
ULONG IOS2_REQ_IO_SIZE ; wire type specific error
ULONG IOS2_WIREERROR ; packet type
ULONG IOS2_PACKETTYPE ; packet type
STRUCT IOS2_SRCADDR, SANA2_MAX_ADDR_BYTES ; source address
STRUCT IOS2_DSTADDR, SANA2_MAX_ADDR_BYTES ; dest address
ULONG IOS2_DATALENGTH ; length of packet data
APTR IOS2_DATA ; packet data (not touched by driver!)
APTR IOS2_STATDATA ; statistics data pointer
APTR IOS2_BUFFERMANAGEMENT ; see SANA-II OpenDevice adoc
LABEL IOS2_SIZE
;
; equates for the IO_FLAGS field
;

SANA2IOB_RAW EQU 7 ; raw packet IO requested
SANA2IOF_RAW EQU (1<<SANA2IOB_RAW)

SANA2IOB_BCAST EQU 6 ; broadcast packet (received)
SANA2IOF_BCAST EQU (1<<SANA2IOB_BCAST)

SANA2IOB_MCAST EQU 5 ; multicast packet (received)
SANA2IOF_MCAST EQU (1<<SANA2IOB_MCAST)

```

```

SANA2IOB_QUICK EQU IOB_QUICK ; quick IO requested (0)
SANA2IOF_QUICK EQU IOF_QUICK

; equates for OpenDevice()
;

SANA2OPB_MINE EQU 0 ; exclusive access requested
SANA2OPF_MINE EQU (1<<SANA2OPB_MINE)

SANA2OPB_PROM EQU 1 ; promiscuous mode requested
SANA2OPF_PROM EQU (1<<SANA2OPB_PROM)

S2_Dummy EQU (TAG_USER+$B0000)
S2_COPYTOBUFF EQU S2_Dummy+1
S2_COPYFROMBUFF EQU S2_Dummy+2

STRUCTURE SANA2DEVICEQUERY,0 ; Standard information
ULONG S2DQ_SIZEAVAILABLE ; bytes available
ULONG S2DQ_SIZESUPPLIED ; bytes supplied
ULONG S2DQ_FORMAT ; this is type 0
ULONG S2DQ_DEVICELEVEL ; this document is level 0
; Common information
UMWORD S2DQ_ADDRFIELDSIZE ; address size in bits
ULONG S2DQ_WTU ; maximum packet data size
ULONG S2DQ_BPS ; line rate (bits/sec)
ULONG S2DQ_HARDWARETYPE ; what the wire is
; Format specific information
LABEL S2DQ_SIZE

; defined SANA-II hardware types
;

S2WIRETYPE_ETHERNET EQU 1
S2WIRETYPE_IEEE802 EQU 6
S2WIRETYPE_ARCNET EQU 7
S2WIRETYPE_LOCALTALK EQU 11
S2WIRETYPE_DYLAN EQU 12
S2WIRETYPE_AMONNET EQU 200
S2WIRETYPE_PPP EQU 253
S2WIRETYPE_SLIP EQU 254
S2WIRETYPE_CSLIP EQU 255

STRUCTURE SANA2PACKETTYPESTATS,0
ULONG S2PTS_TPKNETS ; transmitted count
ULONG S2PTS_RPKNETS ; received count
ULONG S2PTS_TBYTES ; bytes transmitted count
ULONG S2PTS_RBYTES ; bytes received count
ULONG S2PTS_PACKETSDROPPED ; packets dropped count
LABEL S2PTS_SIZE

STRUCTURE SANA2SPECIALSTATRECORD,0
ULONG S2SSR_TYPE ; statistic identifier
ULONG S2SSR_COUNT ; the statistic
APTR S2SSR_STRING ; statistic name
LABEL S2SSR_SIZE

STRUCTURE SANA2SPECIALSTATHEADER,0
ULONG S2SSH_RECORDCOUNTMAX ; room available
ULONG S2SSH_RECORDCOUNTSUPPLIED ; number supplied
LABEL S2SSH_SIZE

STRUCTURE SANA2DEVICESTATS,0

```

```

ULONG S2DS_PACKETSRECEIVED ; received count
ULONG S2DS_PACKETSENT ; transmitted count
ULONG S2DS_BADDATA ; bad packets received
ULONG S2DS_OVERRUNS ; hardware miss count
ULONG S2DS_UNUSED ; currently unused field
ULONG S2DS_UNKNOWNTYPESRECEIVED ; orphan count
ULONG S2DS_RECONFIGURATIONS ; network reconfigurations
STRUCT S2DS_LASTSTART_TV_SIZE ; time of last online
LABEL S2DS_SIZE

; Device Commands
;
S2_START EQU (CMD_NONSTD)
S2_DEVICEQUERY EQU (S2_START+0)
S2_GETSTATIONADDRESS EQU (S2_START+1)
S2_CONFIGINTERFACE EQU (S2_START+2)
S2_ADDMULTICASTADDRESS EQU (S2_START+3)
S2_DELMULTICASTADDRESS EQU (S2_START+4)
S2_MULTICAST EQU (S2_START+5)
S2_BROADCAST EQU (S2_START+6)
S2_TRACKTYPE EQU (S2_START+7)
S2_UNTRACKTYPE EQU (S2_START+8)
S2_GETTYPESTATS EQU (S2_START+9)
S2_GETSPECIALSTATS EQU (S2_START+10)
S2_GETGLOBALSTATS EQU (S2_START+11)
S2_ONEVENT EQU (S2_START+12)
S2_READORPHAN EQU (S2_START+13)
S2_ONLINE EQU (S2_START+14)
S2_OFFLINE EQU (S2_START+15)
S2_END EQU (S2_START+16)

; defined errors for IO_ERROR
;
S2ERR_NO_ERROR EQU 0 ; peachy-keen
S2ERR_NO_RESOURCES EQU 1 ; resource allocation failure
S2ERR_BAD_ARGUMENT EQU 2 ; garbage somewhere
S2ERR_BAD_STATE EQU 3 ; inappropriate state
S2ERR_BAD_ADDRESS EQU 4 ; who?
S2ERR_MTU_EXCEEDED EQU 5 ; too much to chew
S2ERR_NOT_SUPPORTED EQU 6 ; command not supported by hardware
S2ERR_SOFTWARE EQU 7 ; software error detected
S2ERR_OUTOFSERVICE EQU 8 ; driver is offline
;SEE ALSO <exec/errors.h>

; defined errors for IOS2_WIREERROR
;
S2WERR_GENERIC_ERROR EQU 0 ; no specific info available
S2WERR_NOT_CONFIGURED EQU 1 ; unit not configured
S2WERR_UNIT_ONLINE EQU 2 ; unit is currently online
S2WERR_UNIT_OFFLINE EQU 3 ; unit is currently offline
S2WERR_ALREADY_TRACKED EQU 4 ; protocol already tracked
S2WERR_NOT_TRACKED EQU 5 ; protocol not tracked
S2WERR_BUFF_ERROR EQU 6 ; buffer mgmt func returned error
S2WERR_SRC_ADDRESS EQU 7 ; source address problem
S2WERR_DST_ADDRESS EQU 8 ; destination address problem
S2WERR_BAD_BROADCAST EQU 9 ; broadcast address problem
S2WERR_BAD_MULTICAST EQU 10 ; multicast address problem

```

```

S2WERR_MULTICAST_FULL EQU 11 ; multicast address list full
S2WERR_BAD_EVENT EQU 12 ; unsupported event class
S2WERR_BAD_STATDATA EQU 13 ; statdata failed sanity check
S2WERR_IS_CONFIGURED EQU 14 ; attempt to config twice
S2WERR_NULL_POINTER EQU 15 ; null pointer detected
; defined events
;
S2EVENT_ERROR EQU 1 ; error catch all
S2EVENT_TX EQU 2 ; transmitter error catch all
S2EVENT_RX EQU 3 ; receiver error catch all
S2EVENT_ONLINE EQU 4 ; unit is in service
S2EVENT_OFFLINE EQU 8 ; unit is not in service
S2EVENT_BUFF EQU 16 ; buffer mgmt function error catch all
S2EVENT_HARDWARE EQU 64 ; hardware error catch all
S2EVENT_SOFTWARE EQU 128 ; software error catch all
ENDC SANA2_SANA2DEVICE_I

```

```

#define SANA2_SANA2SPECIALSTATS_H
#define SANA2_SANA2SPECIALSTATS_H_1

/*
** $Filename: devices/sana2specialstats.h $
** $Revision: 1.3 $
** $Date: 92/01/10 15:10:29 $
**
** Defined ids for SANA-II special statistics.
**
**
** (C) Copyright 1991 Commodore-Amiga Inc.
** All Rights Reserved
**
*/

#include <sana2/sana2device.h>
#endif /* SANA2_SANA2DEVICE_H */

/*
** The SANA-II special statistic identifier is an unsigned 32
** number. The upper 16 bits identify the type of network wire
** type to which the statistic applies and the lower 16 bits
** identify the particular statistic.
**
** If you desire to add a new statistic identifier, contact
** CATS.
**
*/

/*
** defined ethernet special statistics
**
*/

#define S2SS_ETHERNET_BADMULTICAST (((S2WireType_Ethernet)<0xffff)<<16 | 0x0000)
/*
** This count will record the number of times a received packet
** tripped the hardware's multicast filtering mechanism but was
** not actually in the current multicast table.
**
*/

#define S2SS_ETHERNET_RETRIES (((S2WireType_Ethernet)<0xffff)<<16 | 0x0001)
/*
** This count records the total number of retries which have
** resulted from transmissions on this board.
**
*/

#endif /* SANA2_SANA2SPECIALSTATS_H */

```

```

IFND SANA2_SANA2SPECIALSTATS_I
SANA2_SANA2SPECIALSTATS_I SET 1

/*
** $Filename: devices/sana2specialstats.i $
** $Revision: 1.3 $
** $Date: 92/01/10 15:10:35 $
**
** Defined ids for SANA-II special statistics.
**
**
** (C) Copyright 1991 Commodore-Amiga Inc.
** All Rights Reserved
**
*/

IFND SANA2_SANA2DEVICE_I
INCLUDE "sana2/sana2device.i"
ENDC !SANA2_SANA2DEVICE_I

/*
** The SANA-II special statistic identifier is an unsigned 32
** number. The upper 16 bits identify the type of network wire
** type to which the statistic applies and the lower 16 bits
** identify the particular statistic.
**
** If you'd like to add new statistic identifiers, contact CATS.
**
*/

/*
** defined ethernet special statistics
**
*/

S2SS_ETHERNET_BADMULTICAST EQU (((S2WireType_ETHERNET)<0xffff)<<16 | 00000)
/*
** This count will record the number of times a received packet
** tripped the hardware's multicast filtering mechanism but was
** not actually in the current multicast table.
**
*/

S2SS_ETHERNET_RETRIES EQU (((S2WireType_ETHERNET)<0xffff)<<16 | 00001)
/*
** This count records the total number of retries which have
** resulted from transmissions on this board.
**
*/

ENDC SANA2_SANA2SPECIALSTATS_

```




Envoy

by Greg Miller & Brian Jackson

Background

Networking is one of the fastest growing sections of the computer industry today. In the not-too-distant future, reasonably high speed data connections will be commonplace in most households, providing incredible opportunities for nearly every aspect of computers. Imagine a household computer acting as a video/audio/text answering machine; as a truly *global* news service and encyclopedia; as a platform for amazing multi-player games (consider playing a 3D texture-mapped fantasy role-playing game with thirty of your closest friends -- scattered all over the country, from your *living room!*). Having the ability to connect computers is no longer an expensive option for an elite few -- it is quickly becoming a *necessity*.

The single biggest target of inter-platform networking is communication with Unix machines. AS225, Commodore's TCP/IP package, is intended exactly to fill that need. AS225 supplies standard utilities and features that are mirrors of their Unix equivalents. For the most part, the commands are named the same, act the same, and perform identically to their Unix counterparts. AS225 also supplies an implementation of *sockets*, a standard API for Unix networking applications. Together, these tools provide a reimplement of the Unix networking environment on an Amiga. For many, this is a boon -- programs can be coded in a style familiar to Unix programmers, and recompiled with few modifications to run on the Amiga. However, adopting this type of standard is also detrimental to Amiga programmers. Asking the average Amiga programmer to learn a Unix style of programming for the chore of networking is not practical and tends to discourage application authors from making their products network capable. The overhead involved in providing this kind of compatibility is also daunting -- the size of the support libraries required to do simple communication between machines is considerable.

Envoy, an Amiga peer-to-peer networking package, is designed to eliminate some of these shortcomings. Four of the more important goals in designing *Envoy* were:

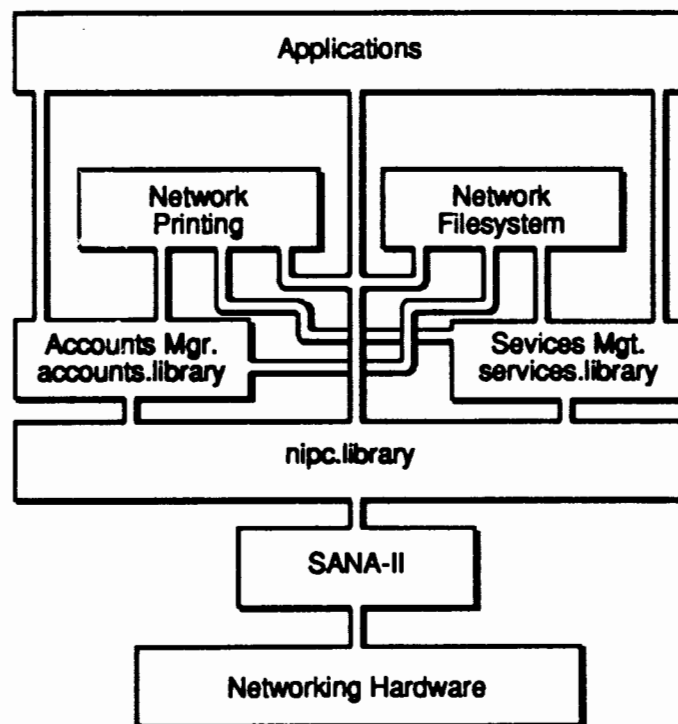
- ☐ Ability to run on low-end Amigas, with as little memory consumption as possible.
- ☐ User-friendly, conforming to Amiga standards for User Interface, style, and operation.
- ☐ Familiar Application Programmer's Interface.
- ☐ Ability to operate over present and future networking hardware, through SANA-II.

This document attempts to provide a technical overview of the operation and structure of Envoy. Envoy alone includes four entirely new shared libraries, a new filesystem, several new devices, and many new configuration tools. Unfortunately, because of the scope of Envoy this document can only serve as a summary.

Architecture

A structural diagram of Envoy is shown in Figure A, where the package as a whole is broken up into several distinctive levels. The lowest two levels include networking hardware and an associated SANA-II driver for that hardware, such as the A2065 ethernet board and the SANA-II a2065.device. A full discussion of SANA-II, a Data Link standard, is available in *SANA-II Network Device Driver Specification - Rev 1.0 23-Apr-92*.

Figure A
Envoy Structural Diagram



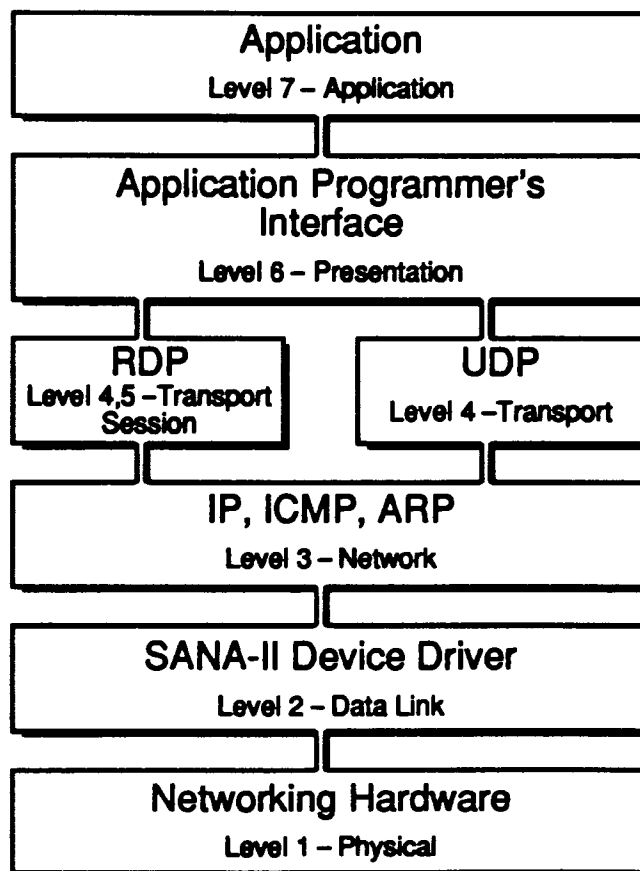
NIPC

Directly above SANA-II lies the communications center of Envoy -- the nipc.library (referred to simply as "NIPC"). NIPC encompasses the problems of communicating with SANA-II devices, fragmenting and defragmenting large amounts of data into smaller pieces, putting all of the pieces it receives in order, dealing with incorrectly received, duplicated, or completely missed packets, routing, subnetting, broadcasting, etc. Essentially, NIPC is an insulator for the applications author from most of the ills of networking.

Implementation

The current implementation of NIPC utilizes several standard protocols to perform the networking tasks needed. Figure B. represents NIPC's architecture.

Figure B
NIPC Architecture



While the current version of NIPC utilizes on a specific protocol stack, including RDP (Reliable Datagram Protocol), UDP (User Datagram Protocol), and IP (Internet Protocol), any side effects of this implementation should not be assumed. Making these kinds of assumptions will damage future compatibility efforts. The following discussion of this NIPC implementation is provided only as a reference.

Figure B. shows the relative layout of NIPC, with reference to the OSI seven-layer model. NIPC occupies levels three through six. When a given application (which conceptually occupies level seven itself) attempts to do network communication, it uses an nipc.library function call --that exists at level six. The API breaks these more abstract requests into specific network actions, then proceeds to utilize either RDP or UDP to fulfill the requests. Both RDP and UDP communicate directly with IP, which breaks up RDP and UDP packets into manageable pieces, and then attempts to guide those pieces to the correct destination. IP then proceeds to utilize a SANA-II device to actually transmit the data. On the destination, this process is reversed.

NIPC's API

NIPC provides an Application Programmer's Interface which is intended to be similar to concepts that Amiga programmers are familiar with. While the API is neither Exec Messages or Exec Devices, it intentionally makes use of certain names and structures of both Devices and Messages.

Understanding several new concepts is vital to understanding NIPC's API. Four of the most important of these concepts are those of *Hostnames*, *Realms*, *Entities* and *Transactions*.

Host Names

Envoy requires that every machine on the network have a unique name that is easily represented in an ASCII string. The name is never interpreted -- only used as a unique identifier for locating that machine. The name should contain only characters (no symbols), be less than 64 characters in total length, and contain no spaces.

Realms

When faced with a network with more than a handful of machines connected, it's useful to break down the machines into more manageable segments. A Realm is merely a conceptual grouping of machines into categories that make sense. For instance, in a company, you might want to group all of the machines on the network depending on which department they're in. For instance, "Marketing", "Sales", "Engineering", and "Product Assurance" are all valid Realm names.

When referencing a machine outside of whatever Realm a user is in, the user must include the Realm name with the host name of the machine he wishes to contact. For instance, if the following people and machines exist on an Envoy network:

Jack, on machine "tofu", in "Marketing"
Terry, on machine "downhill", in "Sales"
Chris, on machine "caffeine", in "Sales"

Since Terry's machine is in "Sales", he can reference Chris's machine by simply typing "caffeine." However, to reference Jack's machine, it's necessary that he indicate that the machine he wants is in a different Realm from his machine. This is done by including the destination Realm name, a colon ":" character, and the destination Hostname. So, for Terry to access Jack's machine, he'd type "Marketing:tofu".

Realms are an advanced feature of Envoy that is only necessary when large numbers of machines are connected to a network. On a simple network with a handful of Amigas, this kind of complexity isn't necessary.

Entities

An Entity is nothing more than a named communications junction. All NIPC communications occurs between two different Entities, with one Entity acting as the source, and one acting as a destination. An Entity can be uniquely identified by:

- ☐ Its name, which, if it exists, is unique on that machine.
- ☐ The name of the machine on which it exists.

Several machines can have Entities that are identically named; however, these identically named Entities exist on *different* machines.

An Entity is created with the CreateEntity() function call. From the nipc.library autdocs:

```
/****** nipc.library/CreateEntityA *****/
*
*  NAME
*    CreateEntityA -- Creates an Entity for communication.
*
*  SYNOPSIS
*    myentity = CreateEntityA(taglist)
*                DO                A0
*
*    struct Entity *CreateEntityA(struct TagItem *);
*    struct Entity *CreateEntity(tag1, tag2, tag3, ...);
*
```

CreateEntity() returns a struct Entity *, which identifies the newly created Entity to NIPC, or NULL for failure. The Entity structure is private, and will not be found in any

public include files. Attempting to access any fields within is illegal, and almost certain to break in the future. Every Entity created must be deleted. This is done with the DeleteEntity() function call.

DeleteEntity() accepts only one parameter -- a pointer to the Entity structure. This pointer can only have been returned by CreateEntity(). From the nipc.library autodocs:

```
/****** nipc.library/DeleteEntity *****/
*
*  NAME
*    DeleteEntity -- Delete an Entity
*
*  SYNOPSIS
*    DeleteEntity(entity);
*                      AO
*
*  VOID DeleteEntity(struct Entity *);
*
```

Every CreateEntity() call must have an associated DeleteEntity() call.

As mentioned above, NIPC communications occur between two different Entities. Before communications can begin, a communications channel must be opened between the two Entities -- in a sense, they must be "connected." This is done with the FindEntity() function call. Given the name of a remote Entity, the name of the machine on which that Entity exists, and an Entity to act as the "source," FindEntity() will attempt to locate the "destination" Entity and create a communications channel between them.

As an example, consider the situation shown in Figure C., where a programmer wishes to use NIPC to communicate between several machines for his ultimate project -- "SuperGame." Two machines are shown, named "Barks" and "Ducks." At the point in time represented by the diagram, "Barks" has three Entities on it -- all of which are owned by different processes. The first Entity on "Barks" is created using something like:

```
ULONG signalbit;
struct Entity *GameEntity;

GameEntity = CreateEntity(ENT_Name, "MyGame's Entity", ENT_PUBLIC,
    TRUE, ENT_AllocSignal, &signalbit,
    TAG_DONE);
```

The Entity on "Ducks" could be similarly created. To communicate between Entity "MyGame's Entity" on machine "Barks" and Entity "Game Server Entity" on machine "Ducks," one of the two Entities must attempt to create a "connection" to the other. In this example, we'll consider the program running on machine "Barks" to be the client, and that running on machine "Ducks" to be the server. Therefore, the client ("Barks") will initiate the connection, e.g., attempt to locate and connect to the server. The program running on "Barks" would need to request a connection between its Entity and that which it expects to find on the server. The information required to do this is:

- ☐ A pointer to the source Entity (returned by CreateEntity() -- in our case, the Entity named "MyGame's Entity").
- ☐ The name of the Entity they wish to connect to.
- ☐ The name of the machine the above Entity exists on.

This could be done by:

```
struct Entity *remoteentity;
ULONG errorcode;

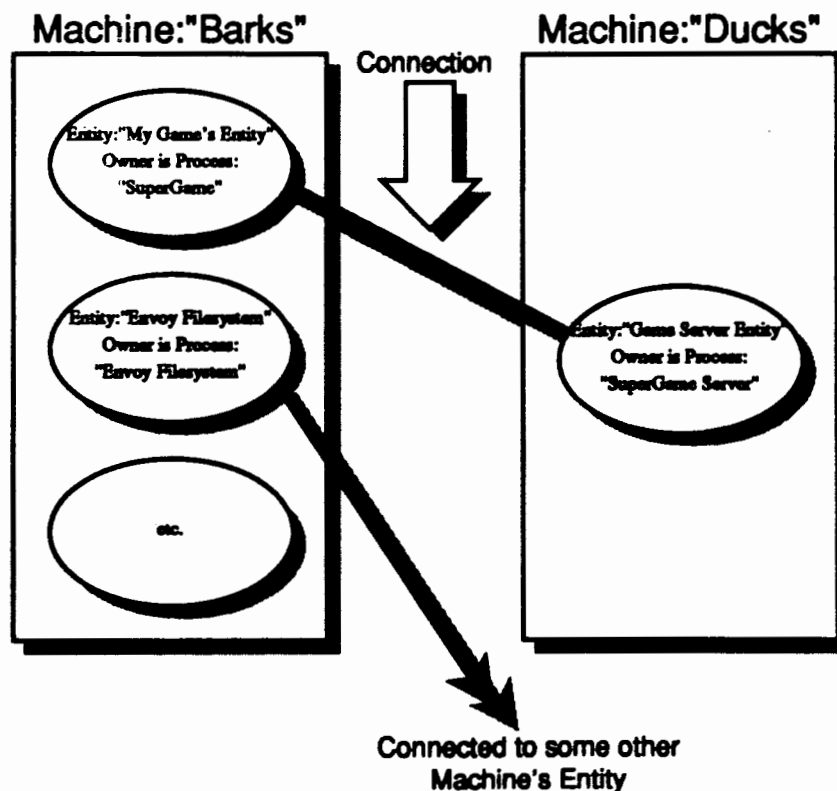
remoteentity = FindEntity("Ducks", "Game Server Entity", GameEntity, &errorcode);

if (!remoteentity)
    printf("Couldn't find the Game's Server!");
```

When the client no longer wishes to maintain the connection between the Entities, it must "shut down" the link between the two created with FindEntity(). This is done by using the LoseEntity() function call on the pointer returned by a successful FindEntity().

Every successful FindEntity() call must have an associated LoseEntity() call.

Figure C
NIPC Communication Between Barks and Ducks



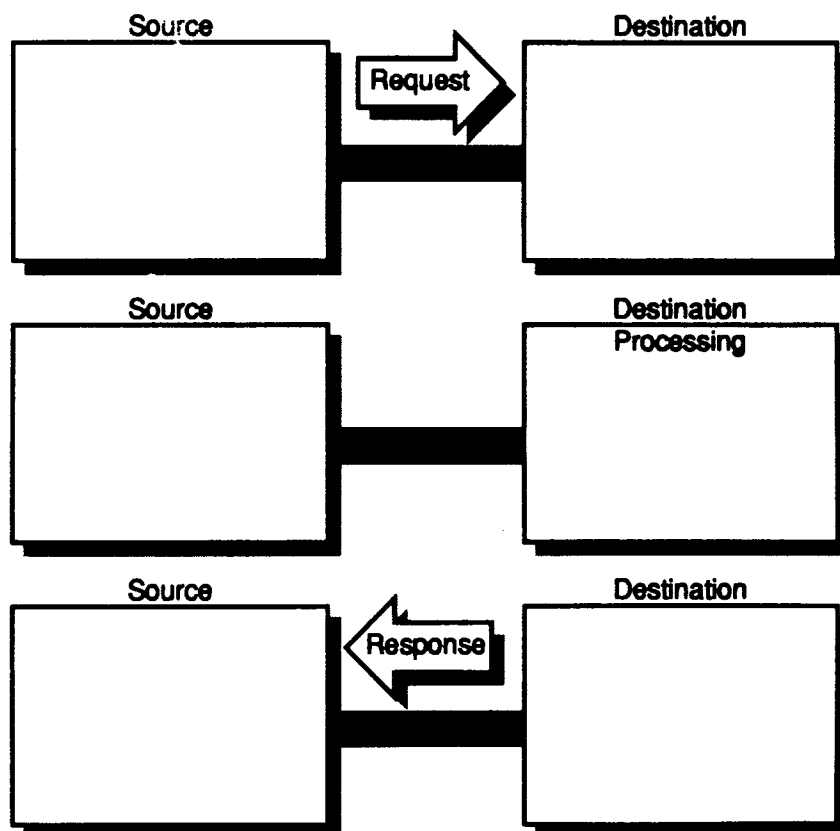
Transactions

The second important concept is the *Transaction*. A Transaction is both the exchange of data in several phases between a pair of Entities, and the name of the system structure used to perform this action. These phases are:

- ❑ The *Request* phase. In this phase, the Transaction travels from the source (or “client”) Entity to the destination (or “server”) Entity, where it waits for the server program to begin to process it.
- ❑ The *Servicing* phase. Between the time the Server program receives the Transaction from the destination Entity, and the time it returns the Transaction to the source, the Transaction is in the Servicing phase.
- ❑ The *Response* phase. At the point where the Transaction is returned from the destination Entity *back* to the source, the Transaction enters the Response phase.

All three phases are shown in Figure D.

Figure D
Transaction Phases



The structure which is always associated with a Transaction is public; however, for future compatibility, all Transactions must be allocated and deallocated with the `AllocTransaction()` and `FreeTransaction()` function calls. From `<envoy/nipc.h>`:

```
struct Transaction
{
    struct Message trans_Msg; /* Used as local carrier */
    struct Entity *trans_SourceEntity; /* T's source */
    struct Entity *trans_DestinationEntity; /* T's destination */
    UBYTE trans_Command; /* Server-proprietary */
    UBYTE trans_Type; /* Explains which phase Transaction is in */
    ULONG trans_Error; /* any error values */
    ULONG trans_Flags; /* see below */
    ULONG trans_Sequence; /* Used by the library to maintain sequences */
    APTR trans_RequestData; /* ptr to request data buffer */
    ULONG trans_ReqDataLength; /* The length of data in the above buffer */
    ULONG trans_ReqDataActual; /* The length of valid data in the request buffer */
    APTR trans_ResponseData; /* ptr to response data buffer */
    ULONG trans_RespDataLength; /* The size of the buffer above */
    ULONG trans_RespDataActual; /* When data returned in a response, amount of the buffer used */
    UWORD trans_Timeout; /* Number of seconds before a timeout */
};
```

All Transactions share the above structure. Being able to transmit and receive Transactions between Entities is useless without some method of attaching user data both at Request time and at Response time. For this reason, a Transaction has two data buffers optionally attached; one buffer for Request data (data sent from the source (or "client")) to the destination (or "server"), and one buffer for Response data (data sent back from the server to the client, as the Transaction returns to its original source). These data areas are user-definable in length as well as structure -- but *must* be allocated before the Transaction is sent. The preferred method for obtaining these buffers is by passing the `TRN_AllocReqBuffer` and `TRN_AllocRespBuffer` tags for `AllocTransaction()`.

However, in consideration of the condition where doing so would require the programmer to do extra copying, it's also possible for the programmer to supply these buffers -- by allocating a Transaction with no buffers with `AllocTransaction()`, then filling in the `trans_RequestData` and `trans_ResponseData` fields to point to the individual buffers, as well as filling in the `trans_ReqDataLength` and `trans_RespDataLength` fields to the lengths of the buffers. When `FreeTransaction()` is called, NIPC will realize that since `AllocTransaction()` didn't allocate those buffers, that they should be freed by the user.

Another important consideration is the condition where the important data in one of the two buffers doesn't entirely fill the buffer. As an example, consider a 1024 byte buffer, where only the first 5 bytes are real data, and the last 1019 bytes are nothing more than garbage. Transmitting the entire 1024 bytes would be horribly wasteful. Therefore, the programmer is required to tell NIPC not only how large the buffers are, but also how much of each buffer contains valid data. Before the client can utilize a Transaction, the programmer is required to provide the length of valid request data in `trans_ReqDataActual`, and before the server is allowed to return a Transaction back to the client, the length of valid response data must be supplied in `trans_RespDataActual`.

Transactions always follow the same conceptual path -- from an Entity on a machine that we'll label the Source (or "client") to an Entity on a machine considered the Destination (or "server"), and back again. Several NIPC function calls provide the capability to perform this kind of operation. Once a Transaction has been created with Request and Response buffers, and the Request buffer has been filled in, the Transaction needs to be transmitted to the destination Entity. This is done either with `BeginTransaction()` or `DoTransaction()` -- which both accept the same parameters. Like `DoIO()`, `DoTransaction()` blocks while waiting for the Transaction to complete. `BeginTransaction()`, like `BeginIO()` or `SendIO()`, starts the Transaction -- and returns immediately, whether the Transaction has completed or not.

Using an analogy of Entities being somewhat similar to Message Ports, and Transactions to Messages (as well as `IORequests`), similarities between functions like `GetMsg()` and `GetTransaction()` should be evident; where `GetMsg()` removes the topmost Message on a Port and returns it, `GetTransaction()` removes the topmost Transaction on an Entity and returns it.

The Request phase of a Transaction, as explained earlier, begins when a Transaction is sent from a source Entity to a destination Entity. Also mentioned above, this is done with either `BeginTransaction()` or `DoTransaction()`. From the autodocs:

```

/***** nipc.library/BeginTransaction *****/
*
*  NAME
*    BeginTransaction -- Start an NIPC Transaction
*
*  SYNOPSIS
*    BeginTransaction(dest_entity,src_entity, transaction)
*                   A0      A1      A2
*
*  VOID BeginTransaction(struct Entity *,struct Entity *,
*                       struct Transaction *);
*

```

Once `BeginTransaction()` or `DoTransaction()` has been used on a Transaction, the Transaction should not be either directly accessed or freed. Only after the Transaction has returned to and is removed from the source Entity should it ever be touched. If asynchronous functionality is important to you, `BeginTransaction()` can be used instead of `DoTransaction()`. If a Transaction arrives at an Entity and if the Entity has a signal bit associated with it, that signal will be set. `WaitEntity()` or `Wait()` can be used to wait for a Transaction to arrive at an Entity.

When a Transaction arrives, it is added to the tail of a FIFO list on the Entity. *The only supported methods for removing a Transaction from an Entity are the functions `GetTransaction()` and `WaitTransaction()`.* `GetTransaction()` removes the top entry in the list, and returns a pointer to that Transaction. If no Transactions are on the given Entity, the function will return `NULL`. `GetTransaction()` is typically used in two cases:

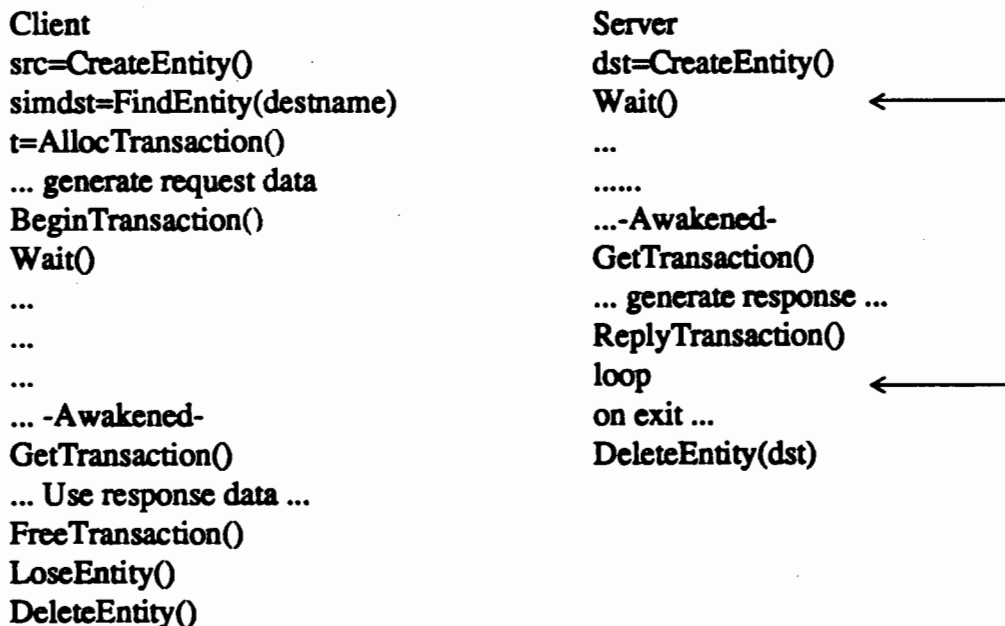
- ☐ Where a client program needs to remove a finished Transaction from the source Entity.

- Where a server program needs to fetch incoming Transactions from a destination Entity.

WaitTransaction() blocks until the given Transaction has completed. When the Transaction completes, WaitTransaction() will wake, remove the given Transaction from the Entity, and return.

Once a server program has finished with a Transaction, it must return the Transaction back to the its source. The only method for doing this is via the ReplyTransaction() function. ReplyTransaction() returns the Transaction and the Response data area back to the source Entity.

Most networking programs, however complex, tend to break down into two programs fulfilling traditional client/server roles. Typical function arrangements for communication between two machines is shown below: *Note: This is provided only as example framework; details such as error checking should be performed in any real code.*



Network Unreliability

The design specification for Envoy states that it offers "reliable data communication." This declaration isn't ideal -- networks in general are completely unreliable. Every networking protocol stack has to deal with the problems of pieces of data never arriving, arriving wrong, arriving a half-hour too late, arriving several times from several sources, and a dozen other

a dozen other problems that all cause immense problems with simply trying to get X bytes from point A to point B. NIPC attempts to deal with these problems in a reasonable manner. However, the possibility always exists that a network will outright fail. Several precautions should be taken by all programmers:

- ☐ *Never* assume a network function succeeded. The fact a network can fail at any place and at random times makes this extremely important.
- ☐ All Transactions should make use of the `trans_Timeout` field. This field indicates a period of time after which NIPC should essentially abort a given Transaction. NIPC takes care in several key places to automatically time out network communications, but without tying up the network with needless *keep-alive* packets, it's entirely possible for one machine to disappear at a point where another machine doesn't notice. The minimum value stored in `trans_Timeout` should be several seconds. Using too great a value can prove annoying to the user, while too small a value can cause software to fail to operate on slow networks. This field should contain the sum of the maximum assumed transfer time for the Transaction (both to the destination, and back from the destination) and the maximum amount of time anticipated for processing and fulfilling a Transaction. A suggested minimum is fifteen seconds.

Other Support

Accounts

Referring again to Figure A., above NIPC lies two support pairs; the Services and the Accounts Managers and libraries.

The Accounts Manager and `accounts.library` provide a bookkeeping service for applications on the notion of user accounts on a specific machine. These accounts are available as an optional layer of security -- to ensure that a user is who they claim to be. It's important to realize that this is only a *layer* of security -- and that no network or system is ever entirely secure. The Accounts Manager is designed to support requests from `accounts.library`, issued on either the same or different machines.

The Accounts system relies heavily on the concept of *Users* and *Groups* to maintain, categorize, and reference different people. A *User* is an accounting entry for a specific individual. Each User has several concepts associated with it:

- ☐ A string representing the owner's name.
Example: "Gregory S. Miller"
- ☐ A password that may optionally be required for use of an account.
Example: "AnyOldPassword"
- ☐ The Primary Group that this user is associated with by default.
Example: "Software Engineering"
- ☐ Several options associated with this User.
Example: Able to create Groups, May Change Password, etc.

A *Group* is a different kind of accounting entry. Rather than being associated with individuals, a Group is associated with collections of individuals. Groups are useful for allowing access to different resources only to certain users. For instance, if a system administrator wished to make a specific hard drive available over the network, but *only* to those involved in DOS, he might create a Group that contained the accounts of those people who were involved in DOS -- and export that hard drive only to the Group called "DOS". A single User may be a member of any number of Groups.

Services

The second of the two support pairs is the Services Manager and `services.library`. The Services mechanism is designed to support starting and running services on demand. A machine can, using Services, export several things, without having to keep the programs that actually perform the exporting chore in memory until that specific service is requested. AS225's `inetd` performs a similar function.

Two types of Services are possible:

- ☐ Those that are *single* invocation. A single program is brought into memory, and this program acts as the server for all remote clients.
- ☐ Those that are *multiple* invocation. Each time a remote client attempts to access the server, a new copy of the server's program is spawned off. Each invocation of the program acts as the server for only one client.

A given service exists as an Amiga shared library, and must provide several standard LVOs, for spawning off new invocations and checking the attributes of a given service.

Not every program that does network communication needs to, or really should be a service. Envoy's Filesystem and Printer sharing facilities are both service-oriented.

However, many applications don't need to be.

The next level shown in Figure A. is defined by the two included examples -- the Network Filesystem and the Network Printer Sharing. These are both no more than included applications. As shown in Figure A., both of these applications have access to all three major Envoy resources available -- NIPC, Services, and Accounts. In fact, both the filesystem and printer sharing applications make use of all three resources. However, not all applications will need to access all three resources. In fact, most will very likely access NIPC alone.

Final Thoughts

Included on the pages following are the autodocs for the `nipc.library`, `services.library`, `accounts.library`, and the `envoy.library`. Although this document is little more than a summary of Envoy, the concepts described will easily allow the development of some truly groundbreaking applications.

Nipc.library Autodocs

TABLE OF CONTENTS

nipc.library/AbortTransaction
 nipc.library/AllocTransactionA
 nipc.library/BeginTransaction
 nipc.library/CheckTransaction
 nipc.library/CreateEntityA
 nipc.library/DeleteEntity
 nipc.library/DoTransaction
 nipc.library/FindEntity
 nipc.library/FreeTransaction
 nipc.library/GetEntityName
 nipc.library/GetHostName
 nipc.library/GetTransaction
 nipc.library/LoadEntity
 nipc.library/NIP_CinquiryA
 nipc.library/PingEntity
 nipc.library/ReplyTransaction
 nipc.library/WaitEntity
 nipc.library/WaitTransaction

nipc.library/AbortTransaction nipc.library/AbortTransaction

NAME

AbortTransaction -- Abort an attempted transaction.

SYNOPSIS

AbortTransaction(transaction)
A1

VOID AbortTransaction(struct Transaction *);

FUNCTION

Aborts a Transaction that was previously started by a call to BeginTransaction().

INPUTS

transaction - Pointer to the Transaction to abort.

RESULT

If the transaction hadn't already been completed (or failed), it will be immediately aborted, and placed on the local Entity.

EXAMPLE

NOTES

Because local Transactions are passed by reference, a transaction sent to a local Entity can be in such a state (being processed) where another process may be referencing them. To ensure that a Transaction is either aborted or completed, an AbortTransaction() call should be followed by a WaitTransaction(). If the transaction given has already completed (or failed), no action will be taken.

BUGS

SEE ALSO

WaitTransaction(), DoTransaction(), BeginTransaction(), CheckTransaction()

nipc.library/AllocTransactionA nipc.library/AllocTransactionA

NAME

AllocTransactionA -- Allocate a transaction structure.

SYNOPSIS

```
transaction = AllocTransactionA(taglist)
DO
    A0
```

```
struct Transaction *AllocTransactionA(struct TagItem *);
struct Transaction *AllocTransaction(tag1, tag2, tag3, ...);
```

FUNCTION

This function will attempt to create a Transaction structure for the user.

INPUTS

taglist -- A pointer to a list of tags defining options and requirements for the structure.

TRN_AllocReqBuffer:

Attempts to allocate a request buffer for you of the size in the ti_Data field of this TagItem. (default: no request buffer)

TRN_AllocRespBuffer:

Attempts to allocate a response buffer for you of the size in the ti_Data field of this TagItem. (default: no response buffer)

A NULL taglist will yield the defaults.

RESULT

transaction -- If conditions permit, AllocTransaction() will return a pointer to a Transaction structure. If the library is unable to either allocate memory or meet the requirements set by the tags passed, the function will return a NULL.

EXAMPLE

NOTES

You are completely free to allocate your own buffer space for either a request or a response buffer. If done, you are also responsible for freeing them.

BUGS

SEE ALSO

nipc.library/BeginTransaction nipc.library/BeginTransaction

NAME

BeginTransaction -- Start an NIPC Transaction

SYNOPSIS

```
BeginTransaction(dest_entity,src_entity, transaction)
A0      A1      A2
```

```
VOID BeginTransaction(struct Entity *struct Entity *,
struct Transaction *);
```

FUNCTION

Attempts to begin a transaction (described appropriately by the transaction structure) to a given entity.

INPUTS

dest_entity - An abstract data type - a "magic cookie" - that identifies the destination entity. (From FindEntity().)

src_entity - An abstract data type - a "magic cookie" - that identifies the destination entity. (From CreateEntity().)

transaction - a pointer to the Transaction to start.

RESULT

None.

EXAMPLE

NOTES

After this function begins, the Transaction structure passed becomes the property of the nipc.library, and CANNOT be freed until this Transaction returns to the source Entity. Once the Transaction returns onto the local Entity, the programmer should check trans_Error, to verify that it completed properly.

Because data transmission isn't an instantaneous operation, a limit must be imposed on exactly how many transactions can be queued up at any given time. Otherwise, it would be for an application program to overflow the throughput of any given networking protocol, occupying all available memory. If the underlying networking protocol becomes overloaded, subsequent calls to BeginTransaction() may block until the networking protocol is no longer overwhelmed.

BUGS

SEE ALSO

DoTransaction(), CheckTransaction(), WaitTransaction(), AbortTransaction(), <envoy/nipc.h>

A NULL taglist will provide the defaults.

RESULT

entity -- a 'magic cookie' that defines your entity, or NULL for failure

EXAMPLE

NOTES

ENT_Signal and ENT_AllocSignal are mutually exclusive.
ENT_Public requires that you also provide ENT_Name; the converse, however, is not true.

BUGS

SEE ALSO

DeleteEntity0

nipc.library/DeleteEntity

NAME

DeleteEntity -- Delete an Entity

SYNOPSIS

DeleteEntity(entity);
A0

VOID DeleteEntity(struct Entity *);

FUNCTION

DeleteEntity0 is the converse of CreateEntity0. It removes the entity from NIPC lists that it may have been attached to, frees up any resources attached to this Entity that were allocated by CreateEntity0, and frees up the actual Entity structure. If NULL is passed as the argument, no action will be taken.

INPUTS

entity -- a 'magic cookie' that defines your Entity

RESULT

None.

EXAMPLE

NOTES

If an attempt is made to DeleteEntity0 an Entity that has been found (via FindEntity0), but not lost (via LoseEntity0), the Entity being deleted will be maintained by NIPC until all references to it no longer exist.
(The same behavior should arguably occur for an Entity on which previously sent Transactions are yet to return.)

IMPORTANT WARNING:

Use DeleteEntity0 ONLY on Entities created with CreateEntity0.
Do NOT use DeleteEntity0 on Entities created with FindEntity0!!

BUGS

CreateEntityA0 LoseEntity0

nipc.library/DoTransaction	nipc.library/DoTransaction
<p>NAME DoTransaction -- Begin a Transaction, and wait for it to complete.</p> <p>SYNOPSIS DoTransaction(dest_entity_src_entity.transaction) A0 A1 A2</p> <p>VOID DoTransaction(struct Entity *, struct Entity *, struct Transaction *);</p> <p>FUNCTION Processes an entire Transaction; it sends the request to the host Entity and awaits the response, or returns at any point because of an error.</p> <p>INPUTS dest_entity - An abstract data type - a "magic cookie" - that identifies the destination entity. (From FindEntity().) src_entity - An abstract data type - a "magic cookie" - that identifies the destination entity. (From CreateEntity().) transaction - a pointer to the Transaction to complete.</p> <p>RESULT None.</p> <p>EXAMPLE</p> <p>NOTES DoTransaction() causes the current context to go into a Wait() state until the Transaction is complete (or cannot be completed. After a DoTransaction() call, the trans_Error field should be checked to verify that the Transaction was properly completed.</p> <p>DoTransaction(), like WaitTransaction() is potentially dangerous, and should be used carefully. If no trans_Timeout value is included, and a server never replies the Transaction (for an unknown reason), DoTransaction() can conceivably wait forever.</p> <p>BUGS</p> <p>SEE ALSO BeginTransaction(), <convoy/nipc.h></p>	<p>nipc.library/FindEntity</p> <p>NAME FindEntity -- Locate a specific entity on a certain host</p> <p>SYNOPSIS remote_entity = FindEntity(hostname.entityname_src_entity.errorptr) D0 A0 A1 A2 A3</p> <p>struct Entity *FindEntity(STRPTR, STRPTR, struct Entity *,ULONG *);</p> <p>FUNCTION Attempts to locate a certain entity on a given host. If you pass a NULL for the entityname, FindEntity will interpret that as "the local machine".</p> <p>INPUTS hostname -- pointer to a null-terminated string declaring the host name of the machine on which you expect to find an Entity named the same as 'entityname'. entityname -- pointer to a null-terminated string declaring the name of a public entity. src_entity -- an Entity returned by CreateEntity() that you wish to use as the 'near' side of a communications path. errorptr -- a pointer to a ULONG in which FindEntity() will provide detailed error information in the event of a FindEntity() failure. If a NULL pointer is passed, no detailed error information will be available.</p> <p>RESULT remote_entity -- NULL if the given Entity cannot be found. Otherwise, the magic cookie describing the Entity that you found.</p> <p>EXAMPLE</p> <p>NOTES If you do a FindEntity to a different machine and it succeeds, you have no guarantee that the remote machine's Entity will continue to exist. Because of this, you should check all returned Transactions for error status - if a Transaction to a remote Entity fails because the nipc.library can no longer find the remote Entity, the transaction will be returned as errored. Regardless of what happens, every SUCCESSFUL FindEntity() REQUIRES an associated LoseEntity().</p> <p>Since the FindEntity() establishes a communications link between the source and destination Entities, it's -dependant- on the source. Therefore, do not DeleteEntity() the source entity before LoseEntity()'ing.</p> <p>BUGS</p>

SEE ALSO
LowEntity0

nipc.library/FreeTransaction	nipc.library/FreeTransaction
NAME	
FreeTransaction -- Free a previously allocated Transaction structure.	
SYNOPSIS	
FreeTransaction(transaction)	
AI	
VOID FreeTransaction(struct Transaction *);	
FUNCTION	
Transaction structures created by AllocTransaction0 (which ought be ALL of them), must be deallocated with FreeTransaction0. FreeTransaction0 frees only the Transaction structure and any portions of the structure that were allocated by the AllocTransaction0 function.	
INPUTS	
transaction - a pointer to the Transaction structure to free if NULL, no action is taken.	
RESULT	
None.	
EXAMPLE	
NOTES	
Don't try to free anything that wasn't allocated by AllocTransaction0.	
BUGS	
SEE ALSO	
AllocTransactionA0	

nipc.library/GetEntityName nipc.library/GetEntityName

NAME
GetEntityName -- Get the ASCII name associated with an Entity.

SYNOPSIS
status = GetEntityName(entity, stringptr, available);
A0 A1 D0

BOOL GetEntityName(STRPTR, STRPTR, ULONG);

FUNCTION
Given a pointer to a string, the length of the string data area, and an entity magic-cookie, this function will attempt to return the name associated with the Entity.

INPUTS
entity - A magic cookie, identifying an Entity.
stringptr - A pointer to a section of memory that you'd like the name copied into.
available - The size of the above string area.

RESULT
status - Either TRUE or FALSE. Note that even if you've referenced an entity, over a network - it's possible that resolving what the correct name is COULD fail.

EXAMPLE

NOTES
If the given Entity has no name, the string "UNNAMED ENTITY" will be returned.

BUGS

SEE ALSO

nipc.library/GetHostName nipc.library/GetHostName

NAME
GetHostName -- Get the name of a specific machine.

SYNOPSIS
status = GetHostName(entity, stringptr, available);
D0 A0 A1 D0

BOOL GetHostName(struct Entity *, STRPTR, ULONG);

FUNCTION
Given a pointer to a string, the length of the string data area, and an Entity magic-cookie, this function will attempt to return the ASCII name associated with the host that the Entity is on.

INPUTS
entity - A magic cookie, identifying an Entity.
(If NULL, the string returned will be either LocalRealm:LocalHost, or simply LocalHost -- depending on whether the local machine is in a realm-based environment or not.)

stringptr - A pointer to a section of memory that you'd like the name copied into.
available - The size of the above string area.

RESULT
status - Either TRUE or FALSE. Note that even if you've referenced an Entity over a network, it's possible that resolving the name COULD fail.

EXAMPLE

NOTES

BUGS

SEE ALSO

nipc.library/GetTransaction nipc.library/GetTransaction

NAME
GetTransaction -- receive the next Transaction waiting on an Entity

SYNOPSIS
transaction = GetTransaction(entity)
DO
 AO

struct Transaction *GetTransaction(struct Entity *);

FUNCTION
GetTransaction() attempts to remove the next available Transaction from the given Entity.

INPUTS
entity -- An entity created by CreateEntity(). (Not from FindEntity() as you're NOT allowed to read from someone else's Entity.)

RESULT
transaction -- NULL, if no Transactions are waiting, a pointer to a Transaction structure otherwise.

EXAMPLE

NOTES
After a successful GetTransaction(), you should check the trans_Type field of a transaction to determine whether it's a request, or a returned failed request/response that was previously BeginTransaction()'d from this Entity.

BUGS

SEE ALSO
BeginTransaction(), DoTransaction(), WaitTransaction(), ReplyTransaction(), <envoy/nipc.h>

nipc.library/LoseEntity nipc.library/LoseEntity

NAME
LoseEntity -- Free up any resources attached from a FindEntity.

SYNOPSIS
LoseEntity(entity)
AO

VOID LoseEntity(struct Entity *);

FUNCTION
This will merely free up any resources allocated with a successful FindEntity() call.

INPUTS
entity -- An abstract data type - a "magic cookie" - that identifies an Entity. NULL for no action.

RESULT
None.

EXAMPLE

NOTES
LoseEntity() should only be used on Entities returned by FindEntity().

BUGS

SEE ALSO
CreateEntity(), DeleteEntity()

nipc.library/NIPCInquiryA nipc.library/NIPCInquiryA

NAME

NIPCInquiryA -- Start a nipc network query

SYNOPSIS

success=NIPCInquiryA(book, maxTime, maxResponses, tagList)
DO A0 D0 D1 A1

BOOL NIPCInquiryA(struct Hook *, ULONG, ULONG, APTR);

success=NIPCInquiry(book, maxTime, maxResponses, firsttag, ...)

BOOL NIPCInquiry(struct Hook *, ULONG, ULONG, Tag, ...);

FUNCTION

Starts an NIPC Network Inquiry. This function can be used to gather data about a single machine, or to gather data for a number of machines. Multiple types of inquiries may be made using this function. Please see below for a description of the types of queries and the types of information you may gather.

The Hook is called for each packet returned from hosts on the network. Depending on how specific your query is, you may or may not get more than one packet. When either maxTime seconds or maxResponses packets have been received, your Hook will be called with a null ParamPckt parameter. If your Hook routine decides that it has all of the information it needs, it may return FALSE. This will cause nipc.library to abort the NIPCInquiry() call.

When the Hook is called, the "Object" parameter will be a pointer to the Task structure for the task or process that called the NIPCInquiryA() function. This is useful if you want to signal the calling task when the query is complete. The Hook "Message" parameter will be a pointer to an array of TagItem structures that contain the query response data from each responding host.

If the query fails for some reason, either due to illegal parameters or to a lack of resources, NIPCInquiryA() will return FALSE. In this case, your Hook function will never be called, so be careful.

INPUTS

book(struct Hook *) - Pointer to a Hook structure to be called for each response.

maxTime(ULONG) - The maximum number of seconds allowed for the query. This is the absolute maximum time allowed.

maxResponses(ULONG) - The maximum number of responses that you will accept from the network.

tagList(struct TagItem *) - Pointer to an array of TagItem structures.

TAGS

The Tags currently defined for NIPCInquiryA() are:

QUERY_IPADDR (ULONG) - This tag is used for querying a machine for its Network IP address. Note: This function is provided purely for diagnostic purposes. Do NOT depend on this tag being available in the future.

MATCH_IPADDR (ULONG) - Query a host with the specified IP address. Please see the note above regarding QUERY_IPADDR.

QUERY_REALMS (STRPTR) - Query a realmserver for the names of all known Realms.

MATCH_REALM (STRPTR) - Only query hosts that are in a specific Realm.

QUERY_HOSTNAME (STRPTR) - Query a host for its name.

MATCH_HOSTNAME (STRPTR) - Only query the host with the specified hostname.

QUERY_SERVICES (STRPTR) - Query a host or hosts for the names of all services on each machine.

MATCH_SERVICE (STRPTR) - Only query hosts that have a specific service specified by the Tag.

QUERY_ENTITIES (STRPTR) - Query a host or hosts for the names of all public entities on each machine.

MATCH_ENTITY (STRPTR) - Query only hosts that have a public entity specified by the Tag.

QUERY_OWNER (STRPTR) - This Tag is currently ignored.

MATCH_OWNER (STRPTR) - This Tag is currently ignored.

QUERY_ATTFLAGS (ULONG) - This Tag allows you to find out what bits are set in ExecBase->AttrFlags.

MATCH_ATTFLAGS (ULONG) - Query only hosts that have the specified bits set in ExecBase->AttrFlags.

QUERY_LIBVERSION (VOID) - This Tag is currently ignored.

MATCH_LIBVERSION (VOID) - This Tag is currently ignored.

QUERY_CHIPREVBITS (ULONG) - This Tag allows you to find out what bits are set in GfxBase->ChipRevBits.

Note: Only the lower 8 bits of the ULONG are used.

MATCH_CHIPREVBITS (ULONG) - Query only hosts that have the specified bits set in GfxBase->ChipRevBits.

QUERY_MAXFASTMEM (ULONG) - Query a machine for it's maximum amount of FAST memory.

MATCH_MAXFASTMEM (ULONG) - Query only hosts with at least the specified amount of FAST memory.

QUERY_AvailFASTMEM (ULONG) - Query a machine for the amount of free FAST memory it has available.

MATCH_AvailFASTMEM (ULONG) - Query only those hosts that have at least the specified amount of FAST memory available.

QUERY_MAXCHIPMEM (ULONG) - Query a machine for it's maximum amount of installed CHIP memory.

MATCH_MAXCHIPMEM (ULONG) - Query only those hosts that have at least the specified amount of CHIP ram installed.

QUERY_AvailCHIPMEM (ULONG) - Query a machine for the amount of free CHIP memory it has available.

MATCH_AvailCHIPMEM (ULONG) - Query only those hosts that have at least the specified amount of CHIP memory available.

QUERY_KICKVERSION (ULONG) - Query a machine for the version and revision of Kickstart it is running.

MATCH_KICKVERSION (ULONG) - Query only those hosts that are running at least the specified version and revision of Kickstart.
 QUERY_WBVERSION (ULONG) - Query a machine for the version and revision of Workbench it is running.
 MATCH_WBVERSION (ULONG) - Query only those hosts that are running at least the specified version and revision of Workbench.

RESULT
 None.

EXAMPLES

Query all hosts in realm "Software" for their Hostname, CPU type, Past memory installed and Kickstart version:

```
NIPInquiry(myhook, /* The hook to call */
2, /* 2 seconds max */
15, /* 15 responses max */
MATCH_REALM,"Software",
QUERY_HOSTNAME,0,
QUERY_ATTFLAGS,0,
QUERY_MAXFASTMEM,0,
QUERY_KICKVERSION,0,
TAG_DONE);
```

The Hook would then get called for each host that responds. An example

TagList passed to the hook might be:

```
MATCH_REALM,"Software",
QUERY_HOSTNAME,"A2500 Test",
QUERY_ATTFLAGS,AFF_68010IAFF_68020IAFF_68881,
QUERY_MAXFASTMEM,4194304, (4 Mega)
QUERY_KICKVERSION,2425007, (VERSION <<16 | REVISION)
TAG_DONE
```

Query a server for all services it has available

```
NIPInquiry(myhook,
2,
MATCH_REALM,"Marketing",
MATCH_HOSTNAME,"Market Server",
QUERY_SERVICES,0,
TAG_DONE)
```

A possible response might be:

```
MATCH_REALM,"Marketing",
MATCH_HOSTNAME,"Market Server",
QUERY_SERVICES,"EnvoyFileSystem",
QUERY_SERVICES,"EnvoyPrinterService",
QUERY_SERVICES,"ConferenceService",
TAG_DONE)
```

NOTES

This function is considered very low-level and is provided for network diagnostic functions. You should probably be using the functions supplied in envoy library to do network queries. These will provide "wrappers" for the most common types of queries.

nipc.library/PingEntity nipc.library/PingEntity

NAME
PingEntity -- Calculate the round-trip time between two Entities

SYNOPSIS
elapsedtime = PingEntity(entity, limit)
D0 A0 D0

ULONG PingEntity(struct Entity *, ULONG);

FUNCTION

INPUTS
entity - A magic cookie, identifying an Entity.
limit - Maximum number of microseconds to wait for a response.

RESULT
elapsedtime -
Total number of microseconds elapsed between attempting to query the Entity, and a response returning. If -1L, no response was received in the given timeout interval. Local Entities will return 0 for elapsed time.

EXAMPLE

NOTES

BUGS

SEE ALSO

nipc.library/ReplyTransaction nipc.library/ReplyTransaction

NAME
ReplyTransaction -- Reply a Transaction.

SYNOPSIS
ReplyTransaction(transaction)
A1

VOID ReplyTransaction(struct Transaction *);

FUNCTION
This causes a Transaction received from another Entity to be returned to the sender. You may return data with the Transaction as well. (See the Transaction structure definition for details.)

INPUTS
transaction -- Pointer to a Transaction structure that was returned by a previous GetTransaction() call.

RESULT
None.

EXAMPLE

NOTES

BUGS

SEE ALSO
GetTransaction()

nipc.library/WaitEntity nipc.library/WaitEntity

NAME
WaitEntity -- Waits for a Transaction to arrive at an Entity.

SYNOPSIS
WaitEntity(localentity)
A0

VOID WaitEntity(struct Entity *);

FUNCTION
WaitEntity0 simply causes the current process to Wait0 until something arrives at the entity.

INPUTS
entity - An abstract data type - a "magic cookie" - that identifies an Entity.

RESULT
None.

EXAMPLE

NOTES
This is potentially dangerous - if nothing ever arrives at this Entity, this function will never return. Unless warranted, the caller should be using Wait0 with a signal mask.

BUGS

SEE ALSO
GetTransaction0

nipc.library/WaitTransaction nipc.library/WaitTransaction

NAME
WaitTransaction -- Waits for a Transaction to complete.

SYNOPSIS
WaitTransaction(transaction)
A1

VOID WaitTransaction(struct Transaction *);

FUNCTION
Waits for the given transaction to complete, or return as failed.

INPUTS
transaction -- a pointer to a Transaction structure.

RESULT
None.

EXAMPLE

NOTES
- When doing networked transactions, if for any reason the remote machine fails to send a response, (but no network problem exists) and you did not supply a timeout value in trans_timeout, WaitTransaction0 can Wait0 forever.
- If you attempt to WaitTransaction0 on a transaction that has already been responded to, WaitTransaction0 will return immediately.

BUGS

SEE ALSO
BeginTransaction0, DoTransaction0, exec.library/Wait0

Envoy.library Autodocs

TABLE OF CONTENTS

envoy.library/HostRequestA
envoy.library/LoginRequestA
envoy.library/UserRequestA

envoy.library/HostRequestA envoy.library/HostRequestA

NAME

HostRequestA -- Create a std. requester for selecting a host

SYNOPSIS

```
ret = HostRequestA(taglist)
DO
    A0
```

```
BOOL HostRequestA(struct TagList *);
BOOL HostRequest(tag1, tag2, ...);
```

FUNCTION

Creates a system requester that allows the user to search for and select the different hosts and realms known on your network.

INPUTS

taglist - Made up of the following possible tags:

HREQ_Buffer - Specifies a pointer to the buffer where you wish the resolved host and/or realm name to be stored when the user selected "OK".

If a given machine exists in a realm, the string returned will be "realmname:hostname".

HREQ_BufferSize - Maximum number of bytes allowed to be copied into the above buffer.

HREQ_Left - Initial left coordinate of the requester.

HREQ_Top - Initial top coordinate of the requester.

HREQ_Width - Horizontal size of requester in pixels.

HREQ_Heights - Vertical size of requester in pixels.

HREQ_DefaultRealm -

Defines the realm to initially search for machines in when the requester first appears. (String should NOT include a ':').

HREQ_NoRealms - Removes the 'realms' button, and prevents users from switching realms by typing "realm:" in the string gadget. The response will only contain a hostname.

HREQ_Screen - Defines the screen on which this requester should be created. If not provided, it will be opened on the workbench screen.

HREQ_Title - Provides the name for the title bar of the requester's window.

HREQ_NoResizer -

Prevents the requester's window from opening with a resizer gadget; the requester will be locked in at the initial size.

HREQ_NoDragBar -

Prevents the requester's window from opening with a dragbar gadget; the requester will be locked in at the original position.

MATCH_... Any of the MATCH tags for

nipc.library/NIPCInquiry() can be included, and will be used to limit hosts that appear to those that meet the given criteria.

RESULT

ret - either TRUE or FALSE, representing whether the requester was successful or not.

EXAMPLE

NOTES

BUGS

SEE ALSO

nipc.library/NIPCInquiry, nipc.library/GetHostName

envoy.library/LoginRequestA

envoy.library/LoginRequestA

NAME

LoginRequestA -- Create a std. requester for name and password

SYNOPSIS

ret = LoginRequestA(taglist)
DO A0

BOOL LoginRequestA(struct TagList *);
BOOL LoginRequest(tag1, tag2, ...);

FUNCTION

Creates a system requester that allows the user to enter his name and password.

INPUTS

taglist - Made up of the following possible tags:

LRBQ_NameBuff - Specifies a pointer to the buffer where you wish the user's name name to be stored when the user selects "OK".

LRBQ_NameBuffLen - Maximum number of bytes allowed to be copied into the above buffer.

LRBQ_PassBuff - Specifies a pointer the buffer where you wish the user's password to be stored when the user selects "OK".

LRBQ_PassBuffLen - Maximum number of bytes allowed to be copied into the above buffer.

LRBQ_Left - Initial left coordinate of the requester.

LRBQ_Top - Initial top coordinate of the requester.

LRBQ_Width - Horizontal size of requester in pixels.

LRBQ_Height - Vertical size of requester in pixels.

LRBQ_Screen - Defines the screen on which this requester should be created. If not provided, it will be opened on the workbench screen.

LRBQ_Title - Provides the name for the title bar of the requester's window.

LRBQ_NoDragBar - Prevents the requester's window from opening with a dragbar gadget; the requester will be locked in at the original position.

RESULT

ret - either TRUE or FALSE, representing whether the requester was successful or not.

EXAMPLE

NOTES

BUGS

Services.library Autodocs

TABLE OF CONTENTS
services.library/FindService
services.library/LoseService

envoy.library/UserRequestA	envoy.library/UserRequestA
NAME UserRequestA -- Create a std. requester for choosing a user.	
SYNOPSIS ret = UserRequestA(taglist) D0 A0	
BOOL UserRequestA(struct TagList *); BOOL UserRequest(tag1, tag2, ...);	
FUNCTION Creates a system requester that allows the user to choose a username from a list of available users on his system.	
INPUTS taglist - Made up of the following possible tags: UGREQ_NameBuff - Specifies a pointer to the buffer where you wish the user's name to be stored when the user selects "OK". UGREQ_NameBuffLen - Maximum number of bytes allowed to be copied into the above buffer. UGREQ_Left - Initial left coordinate of the requester. UGREQ_Top - Initial top coordinate of the requester. UGREQ_Width - Horizontal size of requester in pixels. UGREQ_Height - Vertical size of requester in pixels. UGREQ_Screen - Defines the screen on which this requester should be created. If not provided, it will be opened on the workbench screen. UGREQ_Title - Provides the name for the title bar of the requester's window. UGREQ_NoDragBar - Prevents the requester's window from opening with a dragbar gadget; the requester will be locked in at the original position.	
RESULT ret - either TRUE or FALSE, representing whether the requester was successful or not.	
EXAMPLE	
NOTES	
BUGS	

services.library/FindService services.library/FindService

NAME
FindServiceA -- Connect to a Service.
FindService -- varargs stub for FindServiceA0.

SYNOPSIS
remote_entity = FindServiceA(remotehost, svcname, srentity, taglist);
D0 A0 A1 A2 A3

struct Entity *FindServiceA(STRPTR, STRPTR, struct Entity *,
 struct TagItem *);

remote_entity = FindService(remoteHost, svcname, srentity, tag1, ...);

struct Entity *FindServiceA(STRPTR, STRPTR, struct Entity *,
 Tag, ...);

FUNCTION
Attempts to locate a certain service on a given host.

INPUTS
remotehost - Pointer to a NULL-terminated string that is the name
of the host on which the service you want to use is provided.
NULL implies the local host.
svcname - Pointer to a NULL-terminated string that is the name
of the service you want to connect to.
srentity - an Entity returned by nipc.library/CreateEntity0 that
you wish to use as the 'near' side of the communications path.

TAGS
Tags for use with FindService0:

FSVC_UserName (STRPTR) - Specifies the name of the user who is
trying to use the service. If NULL, or the tag is not specified, the
call will only be able to connect to a service marked as public
on the remote machine.

FSVC_Password (STRPTR) - The password of the user who is trying to
use the service. Only useful in conjunction with FSVC_UserName.

FSVC_Error (ULONG *) - If specified, a pointer to a ULONG in which a
error code describing the failure if there was one.

RESULT
remote_entity - NULL if the given service cannot be found or access to
the service was denied. Otherwise, a magic cookie describing the
service that you found.

NOTES
All of the rules for FindEntity0 apply here as well. Please read the
Autodoc for FindEntity0 for further information.

Each SUCCESSFUL FindService0 REQUIRES an associated LoseService0.

services.library/LoseService services.library/LoseService

NAME
LoseService -- Free up any resources allocated from FindService().

SYNOPSIS
LoseEntity(entity)
AO

VOID LoseEntity(struct Entity *)

FUNCTION
This will merely free up any resources allocated with a successful FindService() call.

INPUTS
entity - A pointer to the Entity to use when communication with the server process.

RESULT
None

NOTES
LoseService() should only be use on entities returned by FindService. Attempting to use LoseService() on entities created by CreateEntity is asking for trouble.

BUGS
None known.

SEE ALSO
FindService()

Accounts.library Autodocs

TABLE OF CONTENTS

accounts.library/AllocGroupInfo
accounts.library/AllocUserInfo
accounts.library/FreeGroupInfo
accounts.library/FreeUserInfo
accounts.library/IDToGroup
accounts.library/IDToUser
accounts.library/MemberOf
accounts.library/NameToGroup
accounts.library/NameToUser
accounts.library/NextGroup
accounts.library/NextMember
accounts.library/NextUser
accounts.library/VerifyUser
accounts.library/IDToUser

accounts.library/AllocGroupInfo accounts.library/AllocGroupInfo

NAME
AllocGroupInfo -- Allocate a GroupInfo structure.

SYNOPSIS
groupinfo = AllocGroupInfo()
DO

struct GroupInfo *AllocGroupInfo(void);

FUNCTION
Allocates a structure for holding group information. You *must* use this function for allocating a GroupInfo structure, or you risk not being compatible with future versions of accounts.library.

INPUTS
None

RESULT
groupinfo - A pointer to a freshly allocated GroupInfo structure, or NULL if not enough memory was available.

NOTES
GroupInfo structures must be free with a call to FreeGroupInfo().

SEE ALSO
FreeGroupInfo()

accounts.library/AllocUserInfo accounts.library/AllocUserInfo

NAME
AllocUserInfo -- Allocate a UserInfo structure.

SYNOPSIS
userinfo = AllocUserInfo()
DO

struct UserInfo *AllocUserInfo(void);

FUNCTION
Allocates a structure for holding user information. You *must* use this function for allocating a UserInfo structure, or you risk not being compatible with future versions of accounts.library.

INPUTS
None

RESULT
userinfo - A pointer to a freshly allocated UserInfo structure, or NULL if not enough memory was available.

NOTES
UserInfo structures must be free with a call to FreeUserInfo().

SEE ALSO
FreeUserInfo()

accounts.library/FreeGroupInfo accounts.library/FreeGroupInfo

NAME
FreeGroupInfo -- Free a GroupInfo structure.

SYNOPSIS
FreeGroupInfo(groupinfo)
AO

VOID FreeGroupInfo (struct GroupInfo *);

FUNCTION
Frees a GroupInfo structure that was allocated with AllocGroupInfo0.

INPUTS
groupinfo - pointer to the GroupInfo structure that you want to free.

RESULT
None

NOTES
Never use this function to free a GroupInfo structure that wasn't allocated with AllocGroupInfo0.

SEE ALSO
AllocGroupInfo0

accounts.library/FreeUserInfo accounts.library/FreeUserInfo

NAME
FreeUserInfo -- Free a UserInfo structure.

SYNOPSIS
FreeUserInfo(userinfo)
AO

VOID FreeUserInfo (struct UserInfo *);

FUNCTION
Frees a UserInfo structure that was allocated with AllocUserInfo0.

INPUTS
userinfo - pointer to the UserInfo structure that you want to free.

RESULT
None

NOTES
Never use this function to free a UserInfo structure that wasn't allocated with AllocUserInfo0.

SEE ALSO
AllocUserInfo0

accounts.library/IDToGroup	accounts.library/IDToUser	accounts.library/IDToUser
<p>NAME IDToGroup - Find a group by GID</p> <p>SYNOPSIS error = IDToGroup(groupid, groupinfo) D0 A0</p> <p>ULONG IDToGroup(UWORD, struct GroupInfo *)</p> <p>FUNCTION This function will try to find a group by his/her UID. If successfully, the GroupInfo will be filled in with the group's name and GID.</p> <p>INPUTS groupid - the 16-bit groupid if the group you are looking for. groupinfo - Pointer to a GroupInfo structure to fill in.</p> <p>RESULTS Returns 0 if the group is a member of the group, or an error. Please see <envoy/errors.h> for possible error codes.</p> <p>SEE ALSO</p>	<p>NAME IDToUser - Find a user by UID</p> <p>SYNOPSIS error = IDToUser(userid, userinfo) D0 A0</p> <p>ULONG IDToUser(UWORD, struct UserInfo *)</p> <p>FUNCTION This function will try to find a user by his/her UID. If successfully, the UserInfo will be filled in with the user's name, UID, primary GID and flags.</p> <p>INPUTS userid - the 16-bit userid if the user you are looking for. userinfo - Pointer to a UserInfo structure to fill in.</p> <p>RESULTS Returns 0 if the user is a member of the group, or an error. Please see <envoy/errors.h> for possible error codes.</p> <p>SEE ALSO</p>	

accounts.library/MemberOf	accounts.library/MemberOf	accounts.library/NameToGroup
<p>NAME MemberOf -- Verify a user's group membership.</p> <p>SYNOPSIS error = MemberOf(group, user) D0 A0 A1</p> <p>ULONG MemberOf(struct GroupInfo *, struct UserInfo *)</p> <p>FUNCTION This function will consult the users and groups database to see if the given user is a member of the given group.</p> <p>INPUTS group - Pointer to a GroupInfo that is filled in with the name of the group or the group's GID. user - Pointer to a UserInfo that is filled in with the name of the user or the user's UID.</p> <p>RESULTS Returns 0 if the user is a member of the group, or an error. Please see <envoy/errors.h> for possible error codes.</p> <p>SEE ALSO</p>		<p>NAME NameToGroup -- Find a group by name.</p> <p>SYNOPSIS error = NameToGroup(groupName, groupinfo) D0 A0 A1</p> <p>ULONG NameToGroup(STRTYPE, struct GroupInfo *)</p> <p>FUNCTION This function will try to find a named group in the groups database and then fill in the GroupInfo structure with the group's name and GID.</p> <p>INPUTS groupName - The name of the group you want information for. groupinfo - A GroupInfo structure to be filled in.</p> <p>RESULTS Returns 0 if the user is a member of the group, or an error. Please see <envoy/errors.h> for possible error codes.</p> <p>SEE ALSO</p>

accounts.library/NameToUser accounts.library/NameToUser

NAME
NameToUser -- Find a user by name.

SYNOPSIS
error = NameToUser(userName, userinfo)
D0 A0 A1

ULONG NameToUser(STRPTR username, struct UserInfo *)

FUNCTION
This function will try to find a user given a user's name. If the user exists in the users database, the UserInfo struct will be filled in with the user's name, UID, primary GID, and flags.

INPUTS
userName - The name of the user you want information for.
userinfo - A UserInfo structure to be filled in.

RESULTS
Returns 0 if the user is a member of the group, or an error.
Please see <envoy/error.h> for possible error codes.

SEE ALSO

accounts.library/NextGroup accounts.library/NextGroup

NAME
NextGroup - Scan through the group database.

SYNOPSIS
error = NextGroup(groupinfo)
D0 A0

ULONG NextGroup(struct GroupInfo *)

FUNCTION
This function is used for building a list of all groups on the system. The first time you call NextGroup, the ui_GroupID of the GroupInfo structure should be filled in with 0.

Each call to NextGroup() will fill in the GroupInfo structure with the next group in the database. When all group's have been listed, NextGroup() will return with ENVOYERR_LASTGROUP.

INPUTS
groupinfo - Pointer to a GroupInfo structure to fill in with the next Group in the database.

RESULTS
Returns 0 until the last group is filled in, and then returns ENVOYERR_LASTGROUP. You *must* check for other possible error conditions as well.

SEE ALSO
NextUser(), NextMember()

accounts.library/NextMember	accounts.library/NextMember	accounts.library/NextUser
<p>NAME NextMember - Scan through the a group's member list.</p> <p>SYNOPSIS error = NextMember(groupinfo, userinfo) D0 A0 A1</p> <p>ULONG NextMember(struct GroupInfo *, struct UserInfo *)</p> <p>FUNCTION This function is used for building a list of all members in a group. The first time you call NextMember, the ui_UserID of the UserInfo structure should be filled in with 0. The GroupInfo structure should contain either the GID or the name of the group that you want to scan. If both are given, the GID will have priority.</p> <p>Each call to NextMember() will fill in the UserInfo structure with the next user in the group. When all users have been listed, NextMember() will return with ENVOYERR_LASTMEMBER.</p> <p>INPUTS groupinfo - Pointer to a GroupInfo structure that is filled in for the group you want to scan. userinfo - Pointer to a UserInfo structure to fill in with the next member of the group.</p> <p>RESULTS Returns 0 until the last group is filled in, and then returns ENVOYERR_LASTMEMBER. You *must* check for other possible error conditions as well.</p> <p>SEE ALSO NextUser(), NextGroup()</p>	<p>NAME NextUser - Scan through the user database.</p> <p>SYNOPSIS error = NextUser(userinfo) D0 A0</p> <p>ULONG NextUser(struct UserInfo *)</p> <p>FUNCTION This function is used for building a list of all user's on the system. The first time you call NextUser, the ui_UserID of the UserInfo structure should be filled in with 0.</p> <p>Each call to NextUser() will fill in the UserInfo structure with the next user in the database. When all user's have been listed, NextUser() will return with ENVOYERR_LASTUSER.</p> <p>INPUTS userinfo - Pointer to a UserInfo structure to fill in with the next User in the database.</p> <p>RESULTS Returns 0 until the last user is filled in, and then returns ENVOYERR_LASTUSER. You *must* check for other possible error conditions as well.</p> <p>SEE ALSO NextGroup(), NextMember()</p>	

accounts.library/VerifyUser accounts.library/VerifyUser

NAME
VerifyUser -- Verify a user's name and password.

SYNOPSIS
error = VerifyUser(username, password, userinfo)
D0 A0 A1 A2

ULONG VerifyUser(STRPTR, STRPTR, struct UserInfo *)

FUNCTION
This function will consult the Users database and check to see if the username and password given are valid. If they are, accounts.library will fill in the UserInfo structure with the user's UID, primary GID, and the flags set for that user.

INPUTS
username - pointer to a user name.
password - pointer to a user's password.
userinfo - pointer to a struct UserInfo.

RESULTS
VerifyUser() will return 0 if the user and password were valid. Otherwise, an error code will be returned. Please see <envoy/errors.h> for possible error codes.

SEE ALSO

accounts.library/IDToUser accounts.library/IDToUser

NAME
IDToUser - Find a user by UID

SYNOPSIS
error = IDToUser(userid, userinfo)
D0 A0

ULONG IDToUser(ULONG, struct UserInfo *)

FUNCTION
This function will try to find a user by his/her UID. If successfully, the UserInfo will be filled in with the user's name, UID, primary GID and flags.

INPUTS
userid - the 16-bit userid if the user you are looking for.
userinfo - Pointer to a UserInfo structure to fill in.

RESULTS
Returns 0 if the user is a member of the group, or an error. Please see <envoy/errors.h> for possible error codes.

SEE ALSO

Accounts.library Autodocs

TABLE OF CONTENTS

xxx.service/AttemptShutdown
xxx.service/GetServiceAttrA
xxx.service/SetServiceAttrA
xxx.service/StartService

xxx.service/AttemptShutdown	xxx.service/AttemptShutdown
NAME AttemptShutdown -- Inform a service that the system will be going down	
SYNOPSIS AttemptShutdown(desc, seconds) A0 D0	
VOID AttemptShutdown(STPTR, ULONG)	
FUNCTION This function provides a means for informing a service as a whole that it should try to shutdown within a specified amount of time.	
INPUTS desc - Pointer to a string that contains a user-friendly reason of reason of why the service needs to be shut down. May be NULL. seconds - The number of seconds until the end of the world. This may be 0 if shutdown needs to be done immediately.	
NOTES This function is provided as a convenience to services such as filesystems that may wish to start refusing new connections and/or flush any buffered data to disk.	

xxx.service/GetServiceAttrsA	xxx.service/GetServiceAttrsA	xxx.service/SetServiceAttrsA
<p>NAME GetServiceAttrsA -- obtain information about the service. GetServiceAttrs -- varargs sub for GetServiceAttrsA.</p> <p>SYNOPSIS GetServiceAttrsA(taglist) A0</p> <p>VOID GetServiceAttrsA(struct TagItem *); GetServiceAttrs(tag1, ...) VOID GetServiceAttrs(Tag, ...);</p> <p>FUNCTION This function provides a method for determining information about the service. Some of this data will be used by Services Manager for configuration. There will also likely be a predefined set of Tags for information such as the names of the connected users, the current load on the service, etc. Service implementations will also be able to define their own custom Tags for diagnostic/statistical purposes.</p> <p>INPUTS taglist - A taglist containing the tags that you want to get the attributes for.</p> <p>TAGS Tags defined for GetServiceAttrsA0: SVCAtrr_Name (STRPTR) - The name of the service you are providing. (Required)</p>		<p>NAME SetServiceAttrsA -- set attributes of a service. SetServiceAttrs -- varargs sub for SetServiceAttrs</p> <p>SYNOPSIS SetServiceAttrsA(taglist) A0</p> <p>VOID SetServiceAttrsA(struct TagItem *); SetServiceAttrs(tag1, ...) VOID SetServiceAttrs(Tag, ...);</p> <p>FUNCTION This function provides a method for setting attributes for a service. For instance, a configuration editor for the service may need to change the name of the service, permissions, etc. This function provides a standardized method for doing so.</p> <p>INPUTS taglist - A list of TagItem structures to be used by the service to modify it's operation.</p> <p>TAGS Tags defined for SetServiceAttrsA0: SVCAtrr_Name (STRPTR) - The name of the service you are providing.</p>

xxx.service/StartService xxx.service/StartService

NAME

StartService -- Accept a new client connection

SYNOPSIS

error = StartService(taglist)
 d0 A0

ULONG StartService(struct TagItem *);

FUNCTION

This function requests that a service take whatever steps are required to initiate a connection with a new client. The service should fill in entityName with the name of the Entity with which you will be accepting client transactions. If you have any kind of problem during startup, return with a non-zero error code. This code will be passed back to the client that called FindService().

INPUTS

taglist - Pointer to an array of TagItem's passed in by the Services Manager.

TAGS

Tags defined for use with StartService():

SSVC_UserName (STRPTR) - The name of the user requesting your service.

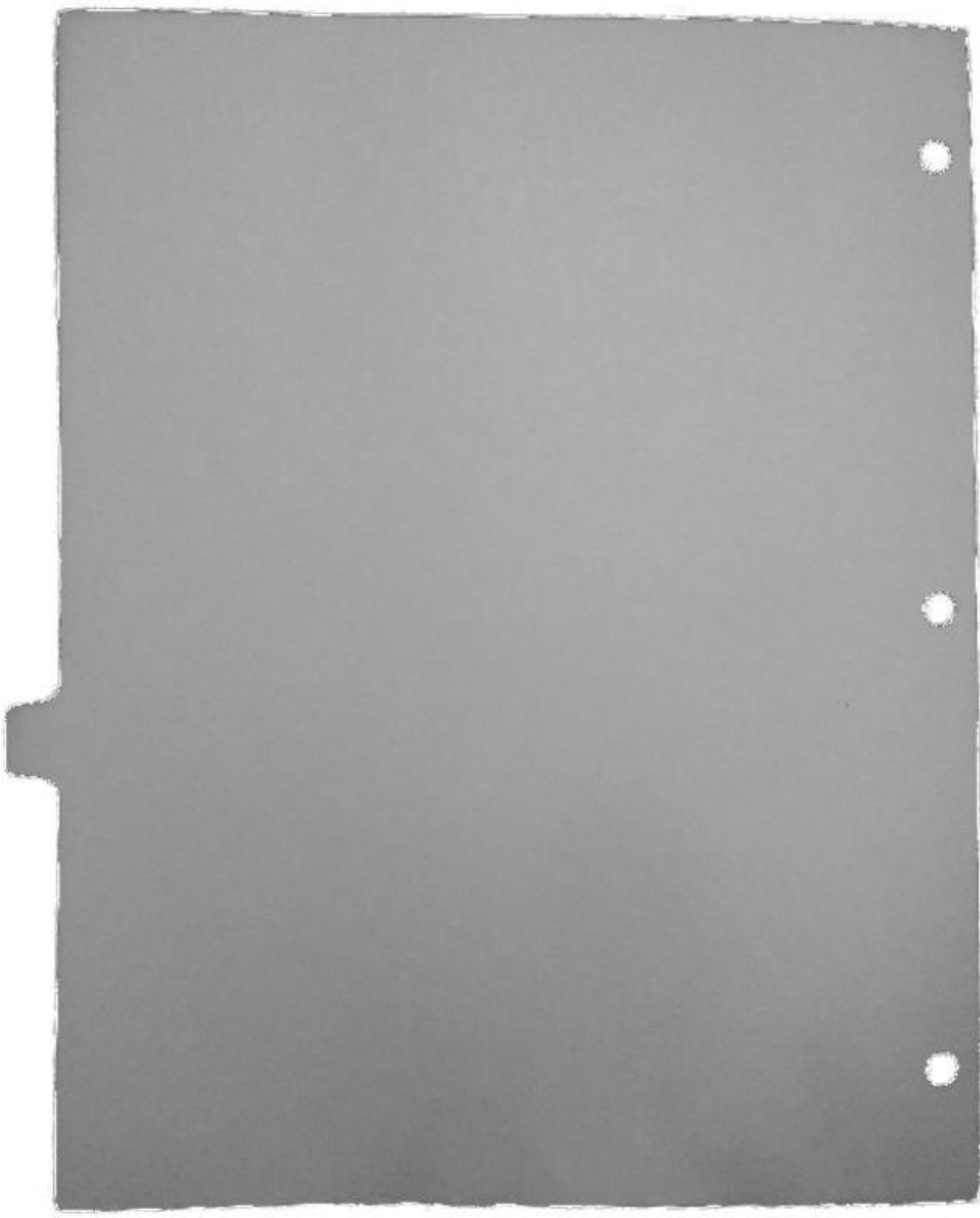
SSVC_Password (STRPTR) - The password of the user requesting your service.

SSVC_HostName (STRPTR) - The hostname of the machine from which the client is connecting.

SSVC_EntityName (STRPTR) - The buffer to fill in with the name of the Entity that the client should connect to.

RESULT

error - A ULONG describing why you could not start your service. This error code will be passed back to the client that called FindService().



An Introduction to Fuzzy Logic and Neural Networks

(With Tutorial Source Code)

By Jim Huffman, Motorola Inc.

Throughout history the model of human thinking has always demonstrated man as capable of two kinds of thought, rational and intuitive. This dual model has impressed itself in the social divisions of science and religion, in psychology via the conscious and subconscious, in biology through the brain structures and proposed specialization of the left and right hemisphere, the nature of the universe in the form of waves and particles or more fundamentally, matter and energy. Now computer technology, long the captive of purely rational models based on sensorially observable cause-effect relations, is being freed to operate in a complementary mode which can be thought of as the superimposition of the model of rationalistic with intuitive thought.

The technologies of artificial neural networks and fuzzy logic merge to provide an alternative to the rational based computational models of Von Neumann. Artificial neural networks and fuzzy logic provide great promise for doing complicated things by freeing the computer from enslavement to sequential algorithmic operation. The new technologies will allow computers to handle approximation and estimation by classifying nonlinear multidimensional control and data space into hyper-geographical areas for neural nets or into subclasses of a continuum of sets known as the universe of discourse in the case of fuzzy logic. Simply stated, artificial neural networks and fuzzy logic work better doing things that computers have stumbled at since birth. The ability to generalize and make decisions based on either pre-existing data relationships or on expert knowledge mean neurals and fuzzy do not necessarily require precise inputs in order to control systems. Thus classical neural and fuzzy operations might be thought of as the "intuitive" side of computing. When used in complementary combination with sequential arithmetic machines, we may have computers in the future which more closely approximate the classical models of human thought.

The programming tools of the future may consist of ways to manipulate input and output data and to capture expert knowledge, then perform a series of experiments to optimize the topology and interconnecting values. Since a lot more "smarts" are going to be in the computer, the computer programmer using these tools will find that systems to be controlled can be looked at in new ways. The methodologies used in applying neural networks and fuzzy logic will re-frame the problem space just the way the invention of tools has always done down through time. When the hammer replaced a hand-held rock, the problems which

could be solved with the old technology were re-framed in the context of the available technology. When the microcomputer was made popular, the problem of control and data manipulation was re-framed. The ability to use a piece of universal computational hardware in various places by simply supplying new programs allowed the construction of machines in high volume. This high-volume production lowered prices so performance could be enhanced and computers could be applied in areas unheard of before their time.

Neural networks and fuzzy logic can be simulated on existing sequential processors, and will provide great advantage in re-framing the problem spaces to which traditional computers have been applied. This means when neural networks or fuzzy logic are applied in a conventional processor, they often provide superior performance over the same hardware applied using conventional algorithmic approaches. The simplification is hidden in the underlying theories of neural and fuzzy operation. The fact that fuzzy, in particular may be implemented on sequential processors with ease leaves processor cycles around for doing those "rational" functions already long associated with microprocessors in control applications. The result is that a microprocessor may be used for conventional control and fuzzy control in the same code space traditionally used just for conventional control.

Are you interested in how it works? Here are the basics: Artificial neural networks (neural networks) have at their root, the goal to simulate biological functions using simple electronic circuits. Conventional computers operate on the commands the programmer supplies in the form of algorithms. Neural networks, on the other hand, "learn" their "programs" from information. Programming a neural network is simply a matter of putting the network into its "learning" mode where a learning rule alters the interconnection strength between the simple electronic circuits. This tends to make the network respond in some desired way to favored stimulus. When the learning is controlled by a template of "correct" answers the network is learning what the teacher wants it to learn. Learning rules determine how much influence is exerted on the network for a given set of training patterns and desired outputs.

To teach the network, we may apply a known stimulus, and present a known reference for the desired response. Ultimately the network will begin to recognize the stimulus-response pairs. With repeated learning the network will learn to recognize the stimulus correctly even in the presence of noise or with partially observable stimulus. If only part of a letter A is presented, the network will recognize the letter A and respond accordingly. If someone makes a letter A that looks a lot like the one that trained the network, a properly trained network will generalize the new model to the letter A it learned. Thus one of the applications of a neural network is in pattern recognition. The neural computer, like the organic brain that inspired it, excels at pattern matching and variations like speech and handwriting recognition. Since the network is also good at guessing what is going to happen next after a series of inputs, it is also good for controlling complicated non-linear processes. We say neural networks are data driven since the application of data via the learning rule is what causes them to "learn" to

operate properly. In other words, the programming of a neural network is controlled by the information that is used to train it. The training information no longer needs to be captured in procedural algorithms impressed on the computer by a programmer.

In practice, an artificial neural network may be constructed to operate on a standard sequential "rational computer" by running a simulation of what would happen in the massively parallel electronic circuits of the ideal neural computer. In such situations each artificial neuron, or neurode, as they are sometimes called, can do a simple process and the processes may be simulated as if they were occurring in parallel as in an ideal neural system. Of course the sequential processor will have to run very fast in order to simulate what a bunch of otherwise simple and slow computation units would do otherwise. In an organic neural computer, the computational elements operate in the millisecond range, but they operate all together to provide a great deal of computational speed and accuracy. When simulating neurons on a Von Neumann machine, you must put up with sharing the single powerful processor over many different processes.

Simply stated, it has been proven useful to create a simple neurode model where the interconnection strength between neurodes is simulated by a multiplier called "weight". The neurode then simply multiplies all its incoming signals by their associated weights and sums them. Thus the action is like the MAC (multiply and accumulate) associated with signal processing. Finally, a non-linear function is performed on the accumulated sum and this becomes the output of the neurode. Thus a neurode has multiple input values and single output values. This is very much like the biological pyramidal neuron cell. The nonlinear function allows multiple layers of neurodes to have significance to each other. It also provides a differentiable function to be used in the famous back propagation training algorithm so common in elementary artificial neural networks. The most famous non-linear function is perhaps the sigmoid function where the output signals are automatically limited from going rail to rail and making the network become a very sophisticated noise generator.

As an appendix to this article, there is a tutorial program written in C. It provides an example neural network which is used to solve a simple temperature control problem. The same problem is used in the example C code for fuzzy logic which follows the neural network example.

Neural networks are concerned with implementing electronic models of actual biological circuitry. Very low level stuff. Fuzzy logic is a relatively new field of mathematics that creates models that behave more like the high level functions of thinking and reasoning. Fuzzy logic allows precise rules to govern operation on imprecise data. Inherently, fuzzy logic finds the most important thing to concentrate on and then uses all its computational power in a focused effort to operate on this simplified picture. Also, fuzzy logic uses a range of operation that covers data conditions between 0 and 1. In other words, it tends to split logical operations into statements which are way more complex than just 0 or 1, the familiar

True and False of Boolean logic. Where neural networks generalize outputs into "guesses" which are percentages of likeness to the training data, fuzzy logic measures the degree of truth of a statement in the beginning, and then applies logical operators to vectors which represent the degrees of truthfulness.

The rules for properness of behavior of a fuzzy system are provided by someone who is an expert on the application. This is someone who may not even be familiar with the underlying physics of a system to be controlled, but may understand what is to be done on a pure experiential level. Thus, the language of fuzzy logic operation uses linguistic descriptors to more easily capture the knowledge of the expert. Since fuzzy logic does not rely on preciseness in its inputs, the linguistic descriptors such as fast, very fast, hot, warm, a little warm, and so on are adequate to describe the data to be operated on. Incoming data may also be in more than one category at a time depending on context. For instance, 75 degrees may be comfortable on a hot day, but cool on a very cold day. The input value did not change, but the way the heating system operates certainly would change depending on the context of the outside temperature.

The process of changing the crisp value of 75 degrees to a fuzzy value is called "fuzzification" and is the first step in fuzzy logic operation. Crisp values are fuzzified by arriving at their truth value over a membership function which is a subset of the series of functions known as the universe of discourse. If the universe of discourse is temperatures on the inside of a car, it may range from a comfort zone of below 50 degrees, to above 90 degrees. These will be broken into membership groups of say COLD, COOL, WARM, and HOT the exact functions and membership dynamics are subjective, determined by the expert description. The shape of the function will determine the truth value for a given input. 75 Degrees may be a member of both COOL and WARM. The degree of membership will determine to what degree the rules that follow apply.

The second step in applying fuzzy logic is the application of the fuzzy rules. The fuzzy rules are simply stated the same as a typical Boolean Rule, except that the operators are fuzzy membership functions. IF the interior of the car is WARM and the outside of the car is COOL, THEN turn the fan on LOW.

Finally, since many rules are liable to apply at any one time, the rules are combined in an output step called defuzzification. Here the combinations of winning rules are converted into a control vector in an appropriate crisp value to control the system. An output might be control voltage in the case of a heater fan.

Fuzzy logic is precise about how it does things. It is only fuzzy about deciding what to do. Because the rules are constant, and the input membership functions the same over the given universe of discourse, the outputs are reliable and repeatable. There is nothing fuzzy about

fuzzy logic in the end. Crisp outputs provide control over the system. If the membership functions are properly shaped and the rules are accurate, then the system can provide a level of control that is unsurpassed for smoothness of operation and reliability.

The appendix contains a fuzzy logic C program tutorial that can be coded into your conventional computer. This is the same heater problem that was used as an example in the neural network tutorial.

You can compile and execute the example programs using most any ANSI C compiler. They will provide output for nearly any machine. I have run them under DOS, Apple OS, and Unix with no problems. The programs are designed as tutorials so you can experiment with inputs, outputs, weights, rules, and so on. They are not the most sophisticated at I/O and depend on simple `getchar()` and `printf()` functions for input and output.

A more sophisticated fuzzy logic training package that is interactive on the IBM PC under Windows is available from your Motorola Sales Office or through your Motorola Semiconductor Products Distributor. There are various configurations of this training package ranging from a 68 dollar introduction, through a 195 dollar more advanced training course, to a several hundreds of dollar package with hardware included.

Once you have mastered the "how it works" part and are ready to put fuzzy logic and neural networks to work for you, you will find many options available to you as an experimenter. The FreeWare line at Motorola (512) 891-FREE contains a fuzzy logic development software package called the KBG, for Knowledge Base Generator. This is a simple fuzzy logic development tool that allows you to create graphical membership functions and write simple linguistic rules for fuzzy logic applications. KBG currently is available only for IBM PCs and clones. The best thing about it is the price. It will cost only the price of the phone call to download it.

For serious development of commercial products or very complicated fuzzy logic systems, Motorola has the Apronix Fuzzy Development Environment - FIDE. It costs under 1,500 dollars, but offers a serious tool which is a must for commercial product development tasks.

There are various neural network development products available. One of the best introductory products we have seen is the NeuralWare Explorer package for IBM PC or Macintosh. It costs just under \$200 the last time I saw a price list. Currently Motorola is working on an interactive course like the fuzzy logic course for neural networks as well. Simple neural development tools are being created to be placed on the FreeWare BBS as well. For serious neural network developing, NeuralWare makes a Professional package. For complicated neural network and for high-speed fuzzy logic applications, only new hardware approaches will work well, in my opinion, and of course that is the area of some considerable research at Motorola.

We have already created a neural network test chip. It operates in pulse mode (like biological neurons) and uses analog storage of weights via charge stored on EEPROM type floating gates. You won't be able to buy one of these chips as it is built strictly for lab experimentation, but from the research we are doing with it, you can expect that a product may be announced in the future.

represent the control surface in this network.

Please mess with these. You can look and see what effect changing weights has on the output of the neuron layers. A lot of fun can be had by making all the weights 64 and putting the correct squashing functions in place. You can put in input values and see the output do its sigmoid shape.

```
/* Second Layer. Weight one. */
LayerTwo_weight1[neuron_1] = 30;
neuron */
LayerTwo_weight1[neuron_2] = 40;
LayerTwo_weight1[neuron_3] = 59;
LayerTwo_weight1[neuron_4] = 55;
LayerTwo_weight2[neuron_1] = 37;
temp */
LayerTwo_weight2[neuron_2] = 40;
LayerTwo_weight2[neuron_3] = 61;
LayerTwo_weight2[neuron_4] = 20;
```

```
/* Third Layer. */
LayerThr_weight[neuron_1] = 30; /* The four inputs to the single output neuron */
LayerThr_weight[neuron_2] = 36;
LayerThr_weight[neuron_3] = 37;
LayerThr_weight[neuron_4] = 61;
```

/*
RUNTIME

This is the crux of the code. This is where the action happens in the network:

- 1). Get the inside and outside temperatures
- 2). Perform normalization into network useable values
- 3). Simulate parallel operation with mult and accumulate
- 4). Convert network output to fan output voltage

```
while(TRUE) /* Loop until next Tuesday */
```

```
/* GET INPUT
```

Get the input temperatures in the ranges indicated. This network likes 0-63 input values so relate that to temperature. So this network will actually work from 50 to 113 degrees. To play with this network as a two input net for general purpose type stuff, you might want to:

- A. Allow input values from 1-64.
 - B. Trap a magic input number and let you alter weights.
 - C. Change the prompts in the printf statements.
 - D. All of the above.
- Reminder: There are some problems with the input going nuts if you don't enter an int.

```
InTemp = OutTemp - 0;
while(InTemp <= 0 || InTemp > 63) {
    printf("Inside Temperature (50 - 100F) = ");
    scanf("%d",&x);
    InTemp = x - 49;
}
```

```
while(OutTemp <= 0 || OutTemp > 63) {
    printf("Outside Temperature (50 - 100F) = ");
    scanf("%d",&x);
    OutTemp = x - 49;
}
```

```
/* FEED THEM NEURODES!
```

```
LayerOne_input[inside] = InTemp;
LayerOne_input[outside] = OutTemp;
```

/* Perform normalization operation in the first layer. This network wants COLDEST to be 63 and HOTTEST to be zero. Subtract values from 63 in layer one of the network.

In the broom balancer I did the 63 - LayerOne_input[outside], but left out the "63 -" part for LayerOne_input[inside]. This made one neuron inhibitory and the other excitatory.

Oh, Oh... you could actually make these layers do product and sums and a sigmoid transfer function if you wanted to have a three layer network. As David Letterman would say, "Have fun kids".

```
LayerOne_output[inside] = 63 - LayerOne_input[inside];
LayerOne_output[outside] = 63 - LayerOne_input[outside];
```

/*
SIMULATE

Pretend like this is a parallel computer architecture by performing the multiplies and adds of weight value times input signal levels one at a time and storing them until you do all the neurodes in a layer. Then do it again until you run out of layers:

```
LayerTwo_output += LayerTwo_weight * LayerOne_output;
```

```
LayerTwo_output[neuron_1] = LayerTwo_weight1[neuron_1] *
LayerOne_output[inside];
LayerTwo_output[neuron_1] += LayerTwo_weight2[neuron_1] *
LayerOne_output[outside];
```

```
LayerTwo_output[neuron_2] = LayerTwo_weight1[neuron_2] *
LayerOne_output[inside];
LayerTwo_output[neuron_2] += LayerTwo_weight2[neuron_2] *
LayerOne_output[outside];
```

```
LayerTwo_output[neuron_3] = LayerTwo_weight1[neuron_3] *
LayerOne_output[inside];
LayerTwo_output[neuron_3] += LayerTwo_weight2[neuron_3] *
LayerOne_output[outside];
```

```
LayerTwo_output[neuron_4] = LayerTwo_weight1[neuron_4] *
LayerOne_output[inside];
LayerTwo_output[neuron_4] += LayerTwo_weight2[neuron_4] *
LayerOne_output[outside];
```

/* Perform the squashing function on the output values. Veggies anyone? */
for (i = 0; i < NOUT; i++)
 Squash(LayerTwo_output[i]);

/* Do layer three (only one neurode in this network) */
LayerThr_output = LayerThr_weight[neuron_1] * LayerTwo_output[neuron_1];
LayerThr_output += LayerThr_weight[neuron_2] * LayerTwo_output[neuron_2];
LayerThr_output += LayerThr_weight[neuron_3] * LayerTwo_output[neuron_3];
LayerThr_output += LayerThr_weight[neuron_4] * LayerTwo_output[neuron_4];

/* Squash and oore the output value. Note, it is customary to use a type 2 in the call for this layer. You may have to do this if you use different values in the network. For now, I like the higher resolution I get with the type 1. Type 1 is range of 0-8K type 2 is 0-16K.

```
/* Squash(LayerThr_output,1);
```

/* Print some stats on the condition of the outputs in each layer. For the input, print the condition of the input layer because the input layer flips the input value over. What you will get is a number between 0 and 63 that will represent the neuron output strength.

```

printf("Layer One   Layer Two   Layer Three\n");
printf(" %2d %2d %2d\n", LayerTwo_output[0], LayerTwo_output[1],
      LayerThr_output);
printf(" %2d %2d %2d\n", LayerTwo_output[2],
      LayerOne_input[1], LayerTwo_output[3]);
/* Normalize the output of the network to 0 - 12 volts to control fan via
   voltage */
volts = LayerThr_output / 5;
LayerThr_output = (int)volts;
printf("Fan voltage is %d volts.\n", LayerThr_output);
/* End the continuous DO loop */
} /* End Main Function */

```

/* Squash function.

A type 1 squash is for the hidden layer neurones. These values are the representation of $2 * 64 * 64$ or 8k of input values squashed to represent the values of 1 - 62.

A type 2 squash is for the output layer which has $4 * 64 * 64$ possibilities or 16k total values to be squashed into 1 - 62 representative values.

If you make fewer layers in your net (broom balancing only takes 3 total neurones) you will have to tweak the divisor in the squashing function to get things to come out right mathematically.

```

Squash(value, type)
  unsigned *value;
  int type;
{
  extern int sigmoid_Freud[64];
  int x;
  if (type == 1)
    x = *value / 128;
  if (type == 2)
    x = *value / 256;
  *value = sigmoid_Freud[x];
} /* End Squash Function */

```

Appendix Two: Fuzzy Logic

```

#define LINT_ARCS
/**** Fuzzy.C *****/

```

LET'S GET FUZZY GANG!

This here fuzzy thing will take an input value and give you an output value, having fuzzified the whole tomato! (or is it potato?)

Anyway, I will do some rules in this code and try to make them obvious. Then I will do a console input for a temperature set inside and outside temperatures. Then I will spit out a Centroid which will happen to be between 0-12 volts to run the fan motor.

File name: Fuzzy.C

Program name: Fuzzifier.prq

Author: Jim Huffman

Date: 10/10/91 (Thanks Bob for all the help.)

Done for IBM PC type machines 10/10/91. Based on the document, "A Brief Description of a Fuzzy Logic Implementation using the Association Engine (AE)" Rev 1.0 Oct 4, 1991, by Bob Seston, Center for Emerging Computer Technologies, Microprocessor and Memory Products Group, Motorola, Inc.
Copyright Motorola, 1991

Description: This is an implementation of the "stuff" in Bob's paper. I thought it would make a good C program and wanted to make something lucid for our customers to look at on Fuzzy Logic implementation this became real.

This guy has 16 rules.

IT - Interior Temp

OT - Outside Temp

FS - Fan Speed.

The rules follow: (subject to change in src test)

Rule #1 IF OT COLD ANND IT COLD THEN FS HIGH

Rule #2 IF OT COLD ANND IT COOL THEN FS HIGH

Rule #3 IF OT COLD ANND IT WARM THEN FS MED

Rule #4 IF OT COLD ANND IT HOT THEN FS LOW

Rule #5 IF OT COOL ANND IT COLD THEN FS HIGH

Rule #6 IF OT COOL ANND IT COOL THEN FS MED

Rule #7 IF OT COOL ANND IT WARM THEN FS LOW

Rule #8 IF OT COOL ANND IT HOT THEN FS OFF

Rule #9 IF OT WARM ANND IT COLD THEN FS MED

Rule #10 IF OT WARM ANND IT COOL THEN FS LOW

Rule #11 IF OT WARM ANND IT WARM THEN FS OFF

Rule #12 IF OT WARM ANND IT HOT THEN FS OFF

Rule #13 IF OT HOT ANND IT COLD THEN FS LOW

Rule #14 IF OT HOT ANND IT COOL THEN FS LOW

Rule #15 IF OT HOT ANND IT WARM THEN FS OFF

Rule #16 IF OT HOT ANND IT HOT THEN FS OFF

```

What you will need to compile this: This thing was done using
LightSpeed's Think C 4.04 on the Macintosh. However, I tried to keep
it pretty ANSI standard.

I used MSC 4.0 to do it on the IBM machine.

.....

/***** includes *****/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

/***** defines *****/
#define NUMRULES24
#define INPUT UNIVERSE45
#define OUTPUT UNIVERSE45
#define MAXARRAY16
#define TRUE1

/***** enums and typedefs *****/
enum { COLD = 0, COOL, WARM, HOT };
enum { OFF = 0, LOW, MED, HIGH };

typedef struct
{
    int start;
    int end;
    int func_array[MAXARRAY];
} MembershipType;

typedef struct
{
    int antecedent1;
    int antecedent2;
} RuleFitType;

/* FUZZYMAIN
Let the Good Times Roll! And good luck trying to unravel all this
cryptography.
*/
main()
{
    int x, /* Index used to count Universe of discourse */
    cnt, /* Index */
    rulecnt, /* Index through the rules */
    classcnt, /* Index through the classes */
    int ConsvVal, AntvVal;

    int InTemp, OutTemp; /* Input temperature and output temperature vars */
    int RuleStrength[NUMRULES]; /* Represents how TRUE the rule is for input values */
    int ConRule[NUMRULES]; /* Which consequent relates to the rule */
    int MaxRuleValue[NUMCLASSES]; /* Stores rule maximums */

    unsigned TotalArea; /* Total area in the Class */
    unsigned MomentArm;
    long DU; /* Center of Moment */

```

```

MembershipType Ant1_Member[NUMCLASSES]; /* Membership classes for antecedents &
consequent */
MembershipType Ant2_Member[NUMCLASSES];
MembershipType Consequent_Member[NUMCLASSES];
RuleFitType Degr_of_Fit[NUMCLASSES]; /* Stores the Degree of fit for antecedents
1 & 2 */

/* SETUP
Setup the membership arrays. This is for the ANTECEDENT1 membership
function. Make triangular shapes the lazy way.
*/
for (x = 0; x < NUMCLASSES; x++)
{
    Ant1_Member[x].startx = x * 10; /* Setup the Rule Membership Arrays */
    Ant1_Member[x].endx = x * 10 + 15; /* Put 0..7..0 in the array */
    for (cnt = 0; cnt < MAXARRAY; cnt++)
    { /* Make each rule overlap by 5 */
        if (cnt < 7)
            Ant1_Member[x].func_array[cnt] = cnt;
        else
            Ant1_Member[x].func_array[cnt] = MAXARRAY - 1 - cnt;
    }
}

/* Setup the membership arrays for the ANTECEDENT2 membership function. */
for (x = 0; x < NUMCLASSES; x++)
{
    Ant2_Member[x].startx = x * 10; /* Setup the Output Membership Arrays */
    Ant2_Member[x].endx = x * 10 + 15; /* Put 0..7..0 in the array */
    for (cnt = 0; cnt < MAXARRAY; cnt++)
    { /* Make each rule overlap */
        if (cnt < 7)
            Ant2_Member[x].func_array[cnt] = cnt;
        else
            Ant2_Member[x].func_array[cnt] = MAXARRAY - 1 - cnt;
    }
}

/* Setup the rule membership arrays for the consequent membership function. */
for (x = 0; x < NUMCLASSES; x++)
{
    Consequent_Member[x].startx = x * 10; /* Setup the Output Membership Arrays */
    Consequent_Member[x].endx = x * 10 + 15; /* Put 0..5, 5..0 in the array */
    for (cnt = 0; cnt < MAXARRAY; cnt++)
    { /* Make each rule overlap */
        if (cnt < 7)
            Consequent_Member[x].func_array[cnt] = cnt;
        else
            Consequent_Member[x].func_array[cnt] = MAXARRAY - 1 - cnt;
    }
}

/* End SETUP */

/* RUNTIME
1). Determine just how TRUE the rule is for the input value.
2). Do the rules
3). Find output membership and center of moment
*/

/*
For a given antecedent1 (inside temperature) and antecedent2 (outside
temperature) determine the degrees of fit and place them in
Degree_of_Fit Array.
*/

```



```

/* End Main */

/*
AAAND(antecedent1, antecedent2): Return Fuzzy AND consequent

Returns the Fuzzy AND of the two integer values. This is just my silly
way to illustrate what is going on in the fuzzy logic so it will be
clear. In a real problem you could hard code the AND or OR functions
and eliminate the overhead of a call. If your logic only used ANDs,
you would take minimum values of the operation. If your logic only
used ORs, you take max.
*/
int AAAND(val1, val2)
int val1, val2;
{
    if (val1 >= val2)
        return(val2);
    else
        return(val1);
} /* End AAAND */

/*
OOR(antecedent1, antecedent2) Returns the Fuzzy OR consequent
*/
int OOR(val1, val2)
int val1, val2;
{
    if (val1 >= val2)
        return(val1);
    else
        return(val2);
} /* End OOR */

```




Writing Applications with AppShell

J. F. Wiederhirn

Oct. 21, 1992

Revised January 11, 1993

1

2

3

Contents

1	Introduction	3
1.1	Intended Audience	3
1.2	Other Sources of Information	3
2	AppShell Overview	4
2.1	Design Goals	4
2.2	Standard User Interface	5
2.3	Object-Oriented Architecture	5
2.4	Infrastructure	6
3	AppShell Foundations	7
3.1	AppShell Kernel	7
3.2	AppShell Message Handlers	8
3.3	AppShell Development Tools	9
3.3.1	AppBuilder	9
3.3.2	AppExchange	9
4	Application Design Concepts	10
4.1	AppShell vs. Normal Amiga Apps	10
4.2	Application as Client	10
4.2.1	Registration Lists	11
4.2.2	Application Code Modules	11
4.3	Application Structure	11
4.3.1	Design Data	11
4.3.2	Init/Shutdown Code Modules	11
4.3.3	Application Code Modules	11
4.3.4	Exception Code Modules	12
5	"Skeleton"	
	A Sample Application	13
5.1	Configuration Data	13
5.2	Application Code Modules	25

A AppShell Kernel Reference	35
B AppShell Handler Reference	38
B.1 AREXX	39
B.2 Command Shell	42
B.3 Intuition (IDCMP)	45
B.4 Notification	60
B.5 Simple Inter-Process Communication (SIPC)	62
B.6 Tool	64
C AppBuilder Design Notes	65
C.1 Overview of AppBuilder's Goals	65
C.2 Overview of AppBuilder's features	66
C.2.1 The Interface Design Editors	66
C.2.2 The Object Librarian	68
C.2.3 The Programmable Source Code Generator	69
C.3 Conclusion	70

Chapter 1

Introduction

AppShell is the core of an object-oriented application framework. It implements many of the basic application support functions which are common in most Amiga applications. For the programmer, this means less time needed to write the "boilerplate" code used by most applications, and more time to concentrate on the sections which make an application distinct.

AppShell provides standard Intuition, console, AREXX and other user interface models to the application. Using AppShell the application gets standard user interface design not only internally, but between all applications using AppShell. The AppShell user interface (UI) elements are all designed to be compliant with the guidelines set for in [Com91d].

1.1 Intended Audience

This manual is intended for anyone interested in developing applications using the AppShell application framework. Some experience with C and basic object-oriented programming concepts is assumed, as well as familiarity with the C development environment on the Amiga.

1.2 Other Sources of Information

Further sources of information include the Amiga ROM Kernel Reference manuals [Com91b, Com91a, Com91c], the Amiga User Interface Style Guide [Com91d], and the documentation for your development environment (such as the SAS/C manuals for their compilers).

Chapter 2

AppShell Overview

Over the last four years the character of the Amiga operating system has undergone some radical changes. The style has changed, many of the user interface and system control elements have changed, as have the appearances and function of most new applications.

The requirements of a standard Amiga application can be overwhelming to a new developer. The initial learning curve to develop on the Amiga is already quite steep, without the additional burden of implementing standard features.

In order to help overcome this initial brick wall, the Amiga AppShell was developed. It provides an overall philosophy on programming applications. As well as providing the mechanisms required to implement the standard Amiga requirements.

2.1 Design Goals

AppShell is heavily based on the M-V-C (Model-View-Controller) scheme of application design. By separating the user interface of the application from the application functions, the application gains a great deal of flexibility. With little work, the application can be controlled by GUI, keyboard/console, scripting language or even another program.

The separation of the user interface from the application functions also gives the user much greater ability to customize the interface to their needs without affecting the functionality of the application. This allows better integration of a set of applications into a unified tool set, as well, which makes complex application environments far more practical.

The design of AppShell was focused on 4 points:

- Easy Design and Prototyping of Applications
- Standardized User Interface Design
- Object Oriented Design and Extensibility
- High User Configurability

The first item, easy design and prototyping, is a serious problem in the Amiga software arena. To date, software design tools have been sparse and limited, and most software is handcrafted. The Amiga's learning curve has always been rather steep, with the recent changes to the system adding yet more for Joe Programmer to know and use.

By providing the "boilerplate" of an application, and using a uniform function call format, algorithmic generation of prototype applications becomes possible. An example of such automation is the AppBuilder program by INOVAtronics. It wraps a nice GUI-based front end around the generation of the AppShell interface specification.

2.2 Standard User Interface

One of the other goals of the AppShell design was to provide a standard toolkit of design elements to use when building a graphical interface. The tools themselves would be able to control their usage, and this would result in an AUSIG [Com91d] compliant interface.

There are many benefits to standardized interface design. It enhances productivity by allowing the user to transfer knowledge from one application to another, without having to learn an entirely new set of tools for each application. One of the given strengths of the Macintosh is a strongly standardized set of UI guidelines, and as far as experiments go, their's has been largely a success.

This is not to say every application has to be precisely the same as the others. By setting the "bottom line" of the interface, and leaving the specifics open to user modification, the application becomes more not less flexible. Also, by preventing tragic aesthetic errors (like accidentally using the Swedish "Free Sex and Beer" symbol on a gadget which starts a hard drive format operation) the user gets a safer application.

2.3 Object-Oriented Architecture

The whole of AppShell was designed with a strong object-oriented philosophy. The message handlers are all expressed as discrete encapsulated units, as is the AppShell kernel itself.

It was a given that application designers would have needs we couldn't predict in AppShell's design. Some designers would also need unique modifications to the "stock" items which wouldn't be viable for the general release. Object-oriented construction is ideal for these situations, because the designer simply replaces the standard handler with one that fits their needs, or writes a new handler to add the needed abilities.

The three main benefits of AppShell's object-orientation are:

- **Code Efficiency.** By using the same library code for many of the common AppShell and handler operations, overall code size can be greatly reduced.
- **Consistent API.** Access to the AppShell and handler functions all use an orthogonal parameter format, and the handlers extend this to also include a "standard" set of tags for the initialization of all handlers.
- **Extensibility.** Advanced users can easily design and add message handlers to supplement the basic AppShell handler selection, perhaps to add new input or output controls, or support new hardware abilities.

2.4 Infrastructure

AppShell is divided into two distinct units: The kernel, and the message handlers. A third segment, the software command bus, exists in the abstract only. The kernel handles operations like library loading, handler opening and closing, and other janitorial operations. The message handlers control the application input/output, hardware access, and any other operations involving things "external" to the application code.

Chapter 3

AppShell Foundations

As stated in Chapter 2, AppShell can be broken into two main components:

- Kernel
- Message Handlers

There are similarities between AppShell's architecture and that of a modern operating system. This is not a coincidence, as many of the concerns of an application framework such as AppShell parallel that of the an OS. What an OS does for access to the raw hardware, application frameworks do for access to the operating system and support routines. It concerns itself with the services an application needs, and mutates the available OS services to fit the application needs using an additional layer of abstraction. In simpler words, it matches the needs of the application with the requirements the OS makes of any program.

One interesting side effect of an extensible application framework such as AppShell is that it gives a quick way to incorporate services until such time as they can be incorporated into the OS. One example of this in AppShell is the SIPC handler, which provides a standard way for the framework to send messages to the application. The OS currently doesn't provide a channel of communication between it and applications in a control sense. This is a feature that has been discussed at great length in the design groups, and is generally seen as a benefit, but difficult to implement for compatibility reasons.

Such features are easily implemented in application frameworks, since you are able to define the ground rules of any applications which use the framework. I fully plan to utilize this particular benefit of AppShell quite a bit in further experimentation (and indeed, some of the network operations such as distributed applications which AppShell makes possible could be *quite* useful).

3.1 AppShell Kernel

The AppShell Kernel operates primarily as the "registration desk" for AppShell applications. It registers the applications, ensures they get the libraries and message handlers they require, and tries to get any optional libraries and handlers as well. It also handles command-line options, icon tooltypes and other features related to the start-up process of the typical application.

Internally, the kernel can be subdivided into two parts: the process management and housekeeping code, and the AppShell handler. The AppShell handler is much closer in nature to the AppShell Message Handlers of the next section, with a similar command interface. It serves as the internal method by which AppShell adds its own management commands to the internal command bus, and how it informs the other message handles and application code modules of changes which affect the application's operation as a whole (such as a panic termination, etc.). This section of code also serves as the liaison between the application and the environment management routines mentioned in the following paragraph.

The process management and housekeeping code is far more transparent to the normal AppShell application. It handles functions like registering launched applications on internal lists (for debugging, among other reasons), providing each application (and instance thereof) with a distinct data storage area, initializing the proper libraries a given application wants and needs, and so forth. Ideally, AppShell applications never interface directly with this code.

The easiest metaphor for the AppShell Kernel in a normal application would be the compiler-specific startup and termination code. Ideally, the application would have no need to interface with it, but because the possibility is there the interfaces are present.

3.2 AppShell Message Handlers

AppShell Message Handlers are the eyes, ears, hands and mouth of the application. They represent the bindings between the application code, the user, and the other sections of the operating system (and by proxy, other applications).

The initial handlers provided with AppShell cover the basic necessities of most applications, but additional niche handlers are easily added to the system. This extensibility allows AppShell to adapt to new hardware and OS features as they are added, and to renovate older handlers when they become outdated or obsolete.

The basic handlers provided with AppShell include:

AmigaGuide Provides context-sensitive interface to AmigaGuide

AREXX Provides general inter-application control interfaces

Command Shell Macro and text-based command entry interface

Intuition Application GUI control

Notification DOS-based file-notification interface

Prefs Application preference interface

SIPC Simple InterProcess Communication interface, think of it as an access port to the internal command bus

Timer Interface to the system timer devices

Tool Asynchronous process-launcher interface

Workbench AppWindow, AppMenu and AppIcon interface

These handlers cover all the basic application interface needs (and in fact, offer quite a bit more than the average hand-coded Amiga application). The AmigaGuide and AREXX interfaces in particular should be implemented by *all* AppShell applications (all applications, AppShell or not, should be making every effort to provide online help and AREXX accessibility).

An application can specify which handlers it needs as a critical part of it's operation, and which others would be convenient but that it could function without having. For example, a database application might require that the AREXX handler be present for it to function, but consider the Intuition handler interface as optional, for use in potential low-memory configurations. AppShell evaluates the application needs at run-time and decides which handlers it can and cannot give to an application.

Ideally, the application will take advantage of as many interfaces as it reasonably can, with the critical and non-critical ones noted as such. Having the widest possible array of input and output connections is generally a positive attribute for an application in the eyes of the user.

3.3 AppShell Development Tools

3.3.1 AppBuilder

AppBuilder, an automated user interface builder, can interface directly with AppShell. In fact, AppBuilder is built on top of AppShell. For more information on AppBuilder, its capabilities, and where to obtain it, please take a look at AppBuilder Design Notes document.

3.3.2 AppExchange

AppExchange is a neat little tool provided with the AppShell examples and developer information. It actually uses information and binding points deep with AppShell to get information on all currently executing AppShell applications. It also binds into each application's command bus to allow you to directly feed commands to them for debugging and testing purposes.

In general, the bindings which AppExchange uses are and will remain private to Commodore. The internal structures involved are *promised* to change frequently and no concessions will be made for applications that do such bindings into AppShell's innards.

Chapter 4

Application Design Concepts

This chapter is an overview of application development for AppShell. It explains the basic issues for application development using AppShell, application behavior under AppShell, and a practical overview of AppShell's components.

For a better overview of the AppShell architecture, you may wish to read Chapter 2 (AppShell Overview). For more information on object-oriented programming and some additional insights into AppShell's design, take a look at [CN91, Cor92].

4.1 AppShell vs. Normal Amiga Apps

The most notable differences between "normal" Amiga applications and AppShell-based applications are never seen by the user. The code reusability of the kernel and handler means the applications themselves are smaller than normal applications (they don't have to carry around all the boilerplate code, it all resides in the kernel and handler). Observant users may notice that the typical AppShell application has a more orthogonal AREXX interface than the average application, a side effect of automatic AREXX support.

The most apparent difference will have to do with the richness of user interface options. Since the cost of adding AREXX interfaces, command shells, AmigaGuide help support, macros and the rest is very small, most AppShell applications will come with all of them. While this seems trivial on paper, the difference in practice can be quite impressive.

4.2 Application as Client

When you "run" an AppShell application, what you're really doing is starting an "AppShell document". The relationship of an application to AppShell is similar to the relationship between a text file and an editor. The application is just a data document which AppShell knows how to properly interpret.

AppShell is constructed to run any number of concurrent applications at once. All get registered on internal lists, and get their own AppShell context space. When an application can have multiple documents, each document also gets its own unique context space (but the application gets registered only a single time on the internal list).

4.2.1 Registration Lists

Registration of the applications with the main AppShell kernel gives some other interesting abilities. The AppExchange tool, for example, is able to “peek” at the internal AppShell lists and provide useful diagnostic information and access to any currently-loaded AppShell application.

Through binding into the kernel and registration lists a different way, remote debugging becomes possible.

4.2.2 Application Code Modules

The actual relationship is a bit more complex, since your compiled code modules are not just interpreted by AppShell. They hang off of links which are part of the application description. When AppShell needs to execute one of your modules, it finds the relevant link and using a standard call method, hands off execution to your code. When your code is complete, it just returns (falling back into the AppShell execution scope).

4.3 Application Structure

All AppShell applications share similar internal design. They can be broken into four types of entities:

- **Design Data**
- **Init and Shutdown Code Modules**
- **Application Code Modules**
- **Exception Code Modules**

4.3.1 Design Data

These are the “instructions” which tell AppShell how to represent your application to the user. They give information on things like the libraries and handlers your application will require, the command set of your application (which also covers the bindings of the application code modules), and the user interface designs which drive the code modules.

4.3.2 Init/Shutdown Code Modules

Compiled code modules, these contain any special code for initialization and shutdown of the application, document, window, or screen. These are provided to easily give a way to bind additional context into an AppShell design, like opening a super-bitmap related to a screen, or such.

4.3.3 Application Code Modules

The core functionality of an AppShell application lives in the application code modules. These are compiled code modules provided by the application designer which handle the “commands” of the application. For example, the code to save and load records in a database, or to recalculate a spreadsheet, or print a document all would reside in application code modules.

4.3.4 Exception Code Modules

Think of exception code modules as disaster preparedness. These provide the panic code to provide clean shutdowns when the world is disintegrating around the application. For example, assertion failure or other foreseeable tragedies would hand execution to an exception code module for a clean escape from the application. The exception code would have the responsibility for removing any special initializations, closing any files, perhaps even attempting to save enough to make a later recovery of the data feasible.

All of these elements combine to form every AppShell application. While there can be tremendous variation of the same elements from application to application, an overwhelming majority will at least have these four types of information.

Chapter 5

“Skeleton” A Sample Application

This chapter is a walk-through of the “Skeleton” sample program. It shows most of the important AppShell features in action, with special emphasis on the new features for V37 AppShell.

5.1 Configuration Data

The program starts like any other C program, with the includes that it needs to compile. There’s nothing particularly unusual here, except that the headers specifically associated with AppShell are included as well.

```
/* skeleton.c
 * Copyright (C) 1991 Commodore-Amiga, Inc.
 * Minimum requirements for an AppShell application.
 * Written by David N. Junod
 * Modified by John F. Wiederhirn
 *
 */

/* Normal Amiga Headers */
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/libraries.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>
#include <graphics/gfx.h>
#include <libraries/gadtools.h>
#include <utility/tagitem.h>
#include <string.h>
```



```

/* Normal Protos and Pragmas */
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/gadtools_protos.h>
#include <clib/utility_protos.h>
#include <pragmas/exec_pragmas.h>
#include <pragmas/intuition_pragmas.h>
#include <pragmas/graphics_pragmas.h>
#include <pragmas/gadtools_pragmas.h>
#include <pragmas/utility_pragmas.h>

/* AppShell-related Headers */
#include <libraries/appshell.h>
#include <clib/appshell_protos.h>
#include <pragmas/appshell_pragmas.h>

```

Following the headers, the BumpRev-generated version information is pulled into the program. AppShell maintains some basic program information around for use in the typical "About..." requester. These defines set up the strings embedded later with that information, and finally the version string is made "real".

```

#include "skeleton_rev.h"

#define APPBASE "SKELETON"
#define APPNAME "Skeleton"
#define APPVERS VERS
#define APPCOPY "Copyright © 1993, Commodore-Amiga, Inc."
#define APPAUTH "David N. Junod & John F. Wiederhirn"

STRPTR ver = VERSTAG;

```

The AppData structure is Skeleton's application-specific data structure. While AppShell doesn't require the data in a single structure, practice has shown that keeping all the data in one place makes determining the memory usage easier. When designing your own applications, you would replace this with any application-specific data your application might need. This structure is nearly always available, with a pointer to it embedded in the "global" AppInfo structure.

```

struct AppData
{
    UBYTE ad_TmpText[128]; /* Temporary text buffer */
    LONG ad_Mode; /* Output mode */
};

```

Every public function in the application has a numeric ID. Many of the standard functions like New, Open, Cut, Quit, etc. have standard ID assignments which are documented in the *libraries/appshell.h* header file.

```
#define DUMMYID APSH_USER_ID
#define CInitID (DUMMYID + 1L)
#define OpenMainID (DUMMYID + 2L)
#define SetModeID (DUMMYID + 3L)
#define ErrorID (DUMMYID + 4L)
#define GetInfoID (DUMMYID + 5L)
#define User1ID (DUMMYID + 6L)
#define User2ID (DUMMYID + 7L)
#define BlurpID (DUMMYID + 8L)
#define PingID (DUMMYID + 9L)
#define LAST_ID (DUMMYID + 11L)
```

Keeping the prototypes for your application functions near the IDs (and the function table) makes it easy to keep track and note discrepancies. All of your application functions need to be prototyped, and note the standardized parameter blocks used. That is the format that AppShell uses when calling the application, and should not be changed.

```
VOID CInitFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID OpenMainFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID SetModeFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID GetInfoFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID ErrorFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID QuitFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID BlurpFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
VOID PingFunc (struct Hook *, struct AppInfo *, struct AppFunction *);
```

This is the function table. It binds the function IDs and function prototypes together. It also sets up the "names" for the functions which are used in the AREXX and SIPC interfaces. Those names are case-insensitive, but in the table should use normal capitalization. The ID is followed by a ReadArgs()-type template and a parameter giving the number of arguments.

```
struct Funcs FTable[] =
{
    {"Quit", QuitFunc, QuitID,},
    {"Error", ErrorFunc, ErrorID,},

    /* Functions with an argument template */
    {"SetMode", SetModeFunc, SetModeID,
     "MALE/S,FEMALE/S,ZOMBIE/S", 3L, NULL,},
    {"GetInfo", GetInfoFunc, GetInfoID, "STEM", 1L, NULL,},

    /* These functions are private, and can not be accessed by the user */
```

```

    {"CInit", CInitFunc, CInitID, NULL, NULL, APSHF_PRIVATE},
    {"OpenMain", OpenMainFunc, OpenMainID, NULL, NULL, APSHF_PRIVATE},

/* Experiment with ARexx scripts */
    {"User1", NO_FUNCTION, User1ID, },
    {"User2", NO_FUNCTION, User2ID, },

/* Test of the winnode delete function */
    {"Blurp", BlurpFunc, BlurpID, },

/* Something to test Pre- and Post- codes */
    {"Ping", PingFunc, PingID, },

/* Marks the end of the array */
    {NULL, NO_FUNCTION, }
};

```

Not all of the possible entries in the Funcs structure are used in the prior example. The entire structure definition is:

```

struct Funcs
{
    UBYTE    *fe_Name;        /* Name of function */
    VOID     (*fe_Func)(struct Hook *, struct AppInfo *,
                        struct AppFunction *);
    ULONG    fe_ID;          /* ID of function */
    STRPTR   fe_Template;    /* Command template */
    ULONG    fe_NumOpts;     /* Number of options in template */
    ULONG    fe_Flags;       /* Status flags for function */
    ULONG    fe_HelpID;      /* Text ID for help string */
                                /* used in command help */
    STRPTR   fe_Params;      /* optional parameters for function */
    ULONG    *fe_GroupID;    /* ^0 terminated array of group ID's */
    LONG     *fe_Options;    /* ReadOnly! ReadArgs */
};

```

The status flags can include values such as APSHF_PRIVATE which tells AppShell that the function is not accessible from external programs or interfaces, such as AREXX or SIPC. The text ID is a Locale string identifier, which corresponds to an item in the application's catalog and default string table. It is used when the user requests help on the function from the command shell (if present). The other parameters (*fe_Params*, *fe_GroupID*, etc.) are rarely used, and beyond the scope of this document.

After defining the functions and function table, Skeleton pulls in the default string table. The string table and the application catalog are both generated by the CatComp developer tool. For more information on CatComp and localization issues in general, consult [Tai91]. The define of CATCOMP_ARRAY indicates to the compiler we want the default strings and IDs pulled in using C array format.

```

/* Here is where the text table used to be, but has been replaced by the */
/* shiny new localized version.                                         */
#define CATCOMP_ARRAY
#include "skeleton_txt.h"

```

One of the functions of the Skeleton demo is to show the new AppShell "screen-jumping" capability by which it can leap from one screen environment to another, even opening and closing screens as it goes. This will be reflected throughout the entire Intuition configuration data by there being duplicates of many of the critical data structures. For example, in the next code blurb there are two separate cycle label arrays and gadget descriptor arrays. In the actual code example, there is only one gadget visible. The second of each structure is used to distinguish it on the second screen. Normally if planning to have a "screen-jumping" application, the gadgets, application, and such would only need a single configuration structure set in the source code. In other words, the duplication here is to make the distinctions more obvious.

The gadget being defined is a cycle gadget. This means it has associated hitbox labels which change between predefined states, and also a gadget label. The `output_data` array below sets up the hitbox labels by giving a (-1) terminated array of text IDs into the application text catalog. The `output_tags` array defines the attributes of the cycle gadget, in this case defining where the gadget's hitbox labels are, and that the gadget label belongs under the gadget itself. For explanation of the definitions, see the Intuition Handler command reference and the header file *libraries/apshattr.h*.

```

LONG output_data[] =
{
    CYCLE_1,
    CYCLE_2,
    CYCLE_3,
    -1
};

struct TagItem output_tags[] =
{
    {APCY_Labels, (ULONG) output_data},
    {APSH_GTFlags, PLACETEXT_BELOW},
    {TAG_DONE, }
};

LONG output_data2[] =
{
    CYCLE_4,
    CYCLE_5,
    CYCLE_6,
    -1
};

struct TagItem output_tags2[] =

```

```

{
{APCY_Labels, (ULONG) output_data2},
{APSH_GTFlags, PLACETEXT_ABOVE},
{TAG_DONE,}
};

```

The Object structure is a building block for AppShell's Intuition interfaces. Even the windows where the gadgets are placed are given Object structures. Object structures congregate in singly-linked lists, where the first element of the structure is a pointer to the next Object in the list (or NULL if it is the last Object in the list).

The Object structure is laid out as follows (from *libraries/appobjects.h*):

```

struct Object
{
    struct Object  *o_NextObject;    /* next object in array */
    WORD           o_Group;          /* Object group */
    WORD           o_Priority;       /* Inclusion priority of object */
    ULONG          o_Type;           /* type */
    ULONG          o_ObjectID;       /* ID */
    ULONG          o_Flags;          /* Object flags */
    UWORD          o_Key;            /* hotkey */
    STRPTR         o_Name;           /* name */
    ULONG          o_LabelID;        /* label index into text catalogue */
    struct IBox    o_Outer;          /* size w/label */
    struct TagItem *o_Tags;          /* tags for object */
    APTR           o_UserData;       /* user data for object */
};

```

Back to the example, in the list of the Object structures, note the Object with the OBJ_Window type. This is the application window. As a side note, you must make sure that the name given to the window in the Object array matches the one given to it in the WindowEnv taglist (explained shortly).

```

struct Object objects[] =
{
    {&objects[1], 0, 0, OBJ_Window, NULL, NULL, NULL, "Main", TEXT_TITLE,
    {0, 0, 200, 50}, NULL,},

    {NULL, 0, 0, OBJ_Cycle, 0, NULL, 0, "Mode", TEXT_LABEL,
    {5, 5, 175, 15}, output_tags,},
};

struct Object objects2[] =
{
    {&objects2[1], 0, 0, OBJ_Window, NULL, NULL, NULL, "Main", TEXT_TITLE2,
    {100, 100, 200, 50}, NULL,},
};

```

```

        {NULL, 0, 0, OBJ_Cycle, 0, NULL, 0, "Mode", TEXT_LABEL,
         {5, 5, 175, 15}, output_tags2,},
};

```

The next part, the ColorSpec, is just a plain Intuition ColorSpec structure. This is used in Skeleton as part of the screen-jumping feature.

```

/* Test NewScreenTags setup */
struct ColorSpec colorspec[] =
{
    {0, 0x000e, 0x000a, 0x000d},
    {1, 0x0000, 0x0000, 0x000f},
    {2, 0x000F, 0x000F, 0x000F},
    {3, 0x000f, 0x0000, 0x0000},
    {-1, 0x0000, 0x0000, 0x0000}
};

struct ColorSpec altcolors[] =
{
    {0, 0x000e, 0x000a, 0x000d},
    {1, 0x0000, 0x000f, 0x0000},
    {2, 0x000F, 0x000F, 0x000F},
    {3, 0x000f, 0x0000, 0x000f},
    {-1, 0x0000, 0x0000, 0x0000}
};

```

And just for fun, and to show a little additional flexibility, the screen used by the Skeleton application will use a different font than the system screen font. In this case, Courier 15 is being used. The definition, a TextAttr structure just goes into the screen tags as normal.

```

/* And a font for luck... */
struct TextAttr dafont = {(STRPTR) "courier.font", 15, 0, 0};

```

Here we have the actual screen definition taglists. These aren't different from screen taglists in any other Amiga application, and all of the parameters have the normal Intuition-defined meanings.

```

UWORD labels1[] = {~0};
struct TagItem stags[] =
{
    {SA_Title, (ULONG) ("Skeleton Screen")},
    {SA_DisplayID, HIRESLACE_KEY},
    {SA_Width, 600},
    {SA_Height, 400},
    {SA_Colors, (ULONG) colorspec},
    {SA_Pens, &labels1},
};

```

```

        {SA_Font, &dafont},
        {SA_Depth, 2},
        {TAG_DONE,}
};

struct TagItem stags2[] =
{
    {SA_Title, (ULONG) ("Skeleton2 Screen")},
    {SA_DisplayID, HIRESLACE_KEY},
    {SA_Width, 640},
    {SA_Height, 400},
    {SA_Colors, (ULONG) altcolors},
    {SA_Pens, &labels1},
    {SA_Font, &dafont},
    {SA_Depth, 2},
    {TAG_DONE,}
};

```

Unlike the screen specifications, the window specifications are unique to AppShell. In combination with the Object structure mentioned earlier, AppShell uses a construct called a WindowEnv to define a given window. For more information on the available tags, check in the description of the Intuition message handler, in the appendices.

The first tag, with the `APSH_NameTag` designator is the name of the Window environment. Next comes a pointer to the list of objects associated with the window. The menus for that window are defined using the `APSH_TTMenu` tag, which means that the menus come defined in the text catalog. The value referred to with the `TEXT_MENU` define is the beginning of the menu definition in the application text catalog.

The `APSH_WinAOpen` tag gives the ID of an application function to be called before AppShell opens that window. This and its complementary tag (not used in the example) `APSH_WinBClose` can be used to attach and remove items like superbitmaps from windows during use. There are also `APSH_WinBOpen` and `APSH_WinBClose` tags for adding things before opening and removing afterwards.

The `APSH_RefreshData` tag gives the ID of the application function to be called when the window needs to be refreshed. For applications which use "painting" in their windows, or other less common situations, this allows the application program to update additional items that AppShell doesn't control.

Finally, the `APSH_CloseWindow` tag tells AppShell what function to call when the close gadget is pressed for that window. In this case, it tells AppShell that the close gadget for the window is telling the application to shutdown.

NOTE: The name used with the `APSH_NameTag` tag must match the one used in the Object entry for that window.

```

/* This is a Window Environment tag list. It describes a window. */
struct TagItem mainenv[] =
{
    {APSH_NameTag, (ULONG) "Main"},
    {APSH_Objects, (ULONG) objects},

```

```

        {APSH_TTMenu, TEXT_MENU},
        {APSH_WinAOpen, OpenMainID},
        {APSH_RefreshData, OpenMainID},
        {APSH_CloseWindow, QuitID},
        {TAG_DONE,}
    };

    struct TagItem mainenv2[] =
    {
        {APSH_NameTag, (ULONG) "Main2"},
        {APSH_Objects, (ULONG) objects2},
        {APSH_TTMenu, TEXT_MENU},
        {APSH_WinAOpen, OpenMainID},
        {APSH_RefreshData, OpenMainID},
        {APSH_CloseWindow, QuitID},
        {TAG_DONE,}
    };

```

The application sets up any libraries it might need by first defining the library bases (including external references to AppShellBase, SysBase, and DOSBase) and then passing the declared bases to AppShell along with tags indicating to AppShell that those libraries need to be opened.

All of the system libraries which the application wants it has designated "required" which means that failure to open them will cause the application to terminate with an error.

The final library requested in the Our_Libs tag array is a non-standard library, and the application has indicated that it can live without it (via the APSH_LibStatus, APSH_OPTIONAL attribute pair.

```

/* Shared system libraries */
extern struct Library *AppShellBase;
extern struct Library *SysBase;
extern struct Library *DOSBase;
struct Library *GadToolsBase;
struct Library *GfxBase;
struct Library *IconBase;
struct Library *IntuitionBase;
struct Library *UtilityBase;
struct Library *MySpecialBase;

/* Library Environment:
 * This tag array is used to open and close the shared system libraries
 * needed by our application.
 */
struct TagItem Our_Libs[] =
{
    /* Minimum library version */

```



```

    {APSH_LibVersion, 36L},

/* All libraries are required */
    {APSH_LibStatus, APSH_REQUIRED},

/* Libraries to open */
    {APSH_GadTools, (ULONG) & GadToolsBase},
    {APSH_Gfx, (ULONG) & GfxBase},
    {APSH_Icon, (ULONG) & IconBase},
    {APSH_Intuition, (ULONG) & IntuitionBase},
    {APSH_Utility, (ULONG) & UtilityBase},

/* The next library is optional */
    {APSH_LibStatus, APSH_OPTIONAL},
    {APSH_LibName, (ULONG) "myspecial.library"},
    {APSH_LibBase, (ULONG) & MySpecialBase},
    {TAG_DONE,}
};

```

The TEMPLATE and OPT_* defines are used to configure the application CLI for launching. AppShell applications all use ReadArgs() to parse the command line arguments.

```

/* Shell argument template */
#define TEMPLATE "Male/S,Female/S,Zombie/S,PubScreen/K,PortName/K"
#define OPT_MALE 0
#define OPT_FEMALE 1
#define OPT_ZOMBIE 2
#define OPT_SCREENNAME 3
#define OPT_PORTNAME 4
#define OPT_STARTUP 5
#define OPT_NOGUI 6
#define OPT_COUNT 7

```

The next few sections of source code represent the configuration tag lists for the various AppShell message handlers that the Skeleton application uses. They tell AppShell how the handlers should be configured at application start, and any information about message handler configuration also tends to hang off of these tag lists.

In the AREXX configuration, the only notable item is the definition of the file extension for the application's AREXX scripts. When the application has AREXX try and resolve a script call, it will tell AREXX to check for files with the ".skel" suffix before checking for any other valid AREXX suffix. In the rating, the APSH_OPTIONAL tag value means that the application can run even though this message handler may not be present.

```

/* ARexx user interface environment specification array */
struct TagItem Handle_AREXX[] =
{
    {APSH_Extens, (ULONG) "skel"},

```

```

        {APSH_Rating, APSH_OPTIONAL},
        {TAG_DONE,}
    };

```

The APSHP_INACTIVE flag in the command shell handler's configuration means that the command shell handler needs to be available, but that the command shell itself should not open until the user explicitly requests it.

```

/* Command Shell user interface environment specification array */
struct TagItem Handle_DOS[] =
{
    {APSH_Status, APSHP_INACTIVE},
    {APSH_Rating, APSH_REQUIRED},
    {TAG_DONE,}
};

```

The Intuition message handler is designated as a required handler. The two other tags indicate that AppShell should call the Ping function after it activates the Intuition interface, and before it shuts the interface back down. There are also complement tags for calling a function before activation and after shutdown of the interface.

```

/* These tags describe the Intuition user interface. */
struct TagItem Handle_IDCMP[] =
{
    {APSH_Rating, APSH_REQUIRED},
    {APSH_DeactiveB, PingID},
    {APSH_ActiveA, PingID},
    {TAG_DONE,}
};

```

The SIPC handler is designated required, as well. Normally, given the low overhead of the SIPC handler, and the potential it has for application control by tools like AppExchange and others, it is a good idea to always make the SIPC interface available.

```

/* These tags describe the Simple IPC user interface. */
struct TagItem Handle_SIPC[] =
{
    {APSH_Rating, APSH_REQUIRED},
    {TAG_DONE,}
};

```

So much for the message handlers. For a more detailed explanation of the options and controls for each of the handlers, please consult the relevant section of the appendix for that handler's command reference.

This next tag list pulls all the previous data structures together and is passed to AppShell. From here, AppShell can determine all it needs to know about a given application to create, manage and terminate it.

Due to the size of this structure, the annotation is going to be mixed in-between the actual data fields instead of at either end of the structure as we did with the previous ones.

The first sections bind in the author and version information mentioned in the very beginning of the chapter. Next, the information about the libraries needed by the application gets bound, and then the function table is added.

```
/* Application Environment:
 * Tell about our application */
struct TagItem Our_App[] =
{
    /* About the application */
    {APSH_AppName, (ULONG) APPNAME},
    {APSH_AppVersion, (ULONG) APPVERS},
    {APSH_AppCopyright, (ULONG) APPCOPY},
    {APSH_AppAuthor, (ULONG) APPAUTH},

    /* Trigger the library opening module */
    {APSH_OpenLibraries, (ULONG) Our_Libs},

    /* Specify the application function table */
    {APSH_FuncTable, (ULONG) FTable},
```

The text information, including a pointer to the default string array pulled in from the *skeleton.txt.h* file, the name of the application catalog, and the language of the default strings (they're in English in this case).

By confining the user data to a single structure, the C `sizeof()` function can be used to determine how much memory AppShell needs to allocate for the application's user data.

For the CLI format and arguments, the previously defined template and argument count are passed in. AppShell provides the results in a `ReadArgs()` structure as a part of the `AppInfo` structure.

```
/* Specify the application text table */
{APSH_DefText, (ULONG) CatCompArray},
{APSH_AppCatalog, (ULONG) "skeleton.catalog"},
{APSH_AppDefLang, (ULONG) "english"},

/* Tell how memory we need for our own data */
{APSH_UserDataSize, sizeof (struct AppData)},

/* Give our Shell startup argument template */
{APSH_Template, (ULONG) TEMPLATE},
{APSH_NumOpts, (ULONG) OPT_COUNT},
```

The message handler configurations are passed in using this structure as well.

```
/* Should always specify the SIPC user interface */
{APSH_AddSIPC_UI, (ULONG) Handle_SIPC},

/* Add an ARexx user interface */
```

```

        {APSH_AddARExx_UI, (ULONG) Handle_AREXX},

/* Add a Command Shell user interface */
        {APSH_AddCmdShell_UI, (ULONG) Handle_DOS},

/* Add an Intuition user interface */
        {APSH_AddIntui_UI, (ULONG) Handle_IDCMP},

```

Finally, to wind up this structure, the application defines a routine which needs to run before AppShell itself starts interpreting things. This routine is given via ID as with the previously mentioned functions.

```

/* Specify a custom initialization routine */
        {APSH_AppInit, CInitID},

        {TAG_DONE,}
};

```

That represents all of Skeleton's static configuration. In actuality, it also represents most of the information involved in designing an application with the exception of the application code itself.

Tools like AppBuilder or others are capable of generating most if not all of this data algorithmically, which makes advanced computer-driven interface design quite possible, and much easier than the old hand-coded methods.

5.2 Application Code Modules

The next section of the Skeleton application consists of the application code modules. Here, the application implements any custom initialization or exit code, the exception code, and the code which represent the functionality of the application. For a database, this would be the code which actually controls the records and files, and handles data gathering (but doesn't actually gather the data, which would fall into the role of the message handlers).

Because the Skeleton application is such a minimalist tool, it doesn't have particularly complex or functional code modules. Some of the modules are a bit contrived in the name of demonstrating a feature without any particular justification. Please keep this in mind when reading, as the code you see will probably not be structured more than grossly similar to Skeleton's code.

The first function, **HandlerFunc()** is nothing more than a convenience. Rather than have to link in a separate support code module, we've just duplicated the basic "varargs parameters in, registerizable parameters out" stub for the routine. This allows the code to use the cleaner notation where the tag list is actually part of the call parameters).

```

BOOL
HandlerFunc (struct AppInfo * ai, ULONG tags,...)
{
    return (HandlerFuncA (ai, (struct TagItem *) & tags));
}

```

The **CInitFunc()** function handles all the pre-initialization tasks. It sets up the cycle gadget state machine, and then launches the Intuition message handler, handing in the Window and Screen environments.

```
VOID CInitFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    extern struct TagItem mainenv[];
    struct AppData *ad = ai->ai_UserData;

    /* See if any Shell arguments were specified */
    if (ai->ai_Options[OPT_MALE])
    {
        ad->ad_Mode = 0L;
    }
    else if (ai->ai_Options[OPT_FEMALE])
    {
        ad->ad_Mode = 1L;
    }
    else if (ai->ai_Options[OPT_ZOMBIE])
    {
        ad->ad_Mode = 2L;
    }

    /* Open the Main window */
    HandlerFunc (ai,
        APSH_Handler, "IDCMP",
        APSH_Command, APSH_MH_OPEN,
        APSH_WindowEnv, (ULONG) mainenv,
        APSH_NewScreenTags, (ULONG) stags,
        TAG_DONE);
}
```

As mentioned earlier, the **OpenMainFunc()** function gets called every time the "Main" window is opened. To simplify the example code, it also gets called every time the window needs to be refreshed, but it could as easily be done using a separate function.

```
VOID
OpenMainFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    struct AppData *ad = (struct AppData *) ai->ai_UserData;
    struct Window *win;
    struct Gadget *gad;

    /* Update the mode gadget */
    if (APSHGetGadgetInfo (ai,
        "Main",
```

```

        "Mode",
        (ULONG *) & win,
        (ULONG *) & gad))
    {
        GT_SetGadgetAttrs (gad, win, NULL, GTCY_Active, ad->ad_Mode, TAG_DONE);
    }
}

```

When the user clicks on the cycle gadget in Skeleton, the `SetModeFunc()` routine gets called. The state machine which keeps track of the gadtool gadget gets updated to the new gadget value.

This routine is also callable via the AREXX, Command Shell or SIPC interfaces. In that context, a new value for the cycle gadget gets passed in, and the gadget state machine (as well as the visual imagery) gets updated to the new value.

```

VOID
SetModeFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    struct AppData *ad = (struct AppData *) ai->ai_UserData;
    struct TagItem *attrs = af->af_Attrs;
    struct TagItem *tag;
    struct Gadget *gad;
    struct Window *win;
    LONG mode = (-1);
    struct Funcs *f;

    if (f = af->af_FE)
    {
        if (f->fe_Options[0])
        {
            ad->ad_Mode = mode = 0;
        }
        else if (f->fe_Options[1])
        {
            ad->ad_Mode = mode = 1;
        }
        else if (f->fe_Options[2])
        {
            ad->ad_Mode = mode = 2;
        }
    }
    /* Being set from the gadget */
    else if (tag = FindTagItem (APSH_MsgCode, attrs))
    {
        ad->ad_Mode = tag->ti_Data;
    }
}

```

```

    }

    /* Update the gadget */
    if ((mode >= 0) &&
        (APSHGetGadgetInfo (ai,
                             "Main",
                             "Mode",
                             (ULONG *) & win,
                             (ULONG *) & gad)))
    {
        /* Update the string gadget */
        GT_SetGadgetAttrs (gad, win, NULL, GTCY_Active, mode, TAG_DONE);
    }
}

```

The **ErrorFunc()** is an example of an exception routine. In this case, to make the function obvious, it is callable from the menu, but the approach it uses to generate the error condition is equally applicable to other error-handling conditions.

```

VOID
ErrorFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    STRPTR name = "Uh Oh!";

    /* sample error return */
    ai->ai_Pri_Ret = RETURN_ERROR;
    ai->ai_Sec_Ret = ERROR_DISPLAY;
    ai->ai_TextRtn = PrepText (ai,
                              APSH_USER_ID,
                              ai->ai_Sec_Ret,
                              (ULONG) name);
}

```

The next routine, **PubScreenName()** is a non-bound application function in that AppShell never calls it directly. Instead its your typical garden variety C function.

```

VOID
PubScreenName (struct Screen * scr, STRPTR buffer)
{
    struct Screen *cs = NULL;
    struct List *publist;
    struct List copy_publist;
    struct PubScreenNode *psnode;
    struct PubScreenNode *copy_psnode;
}

```

```

/* Initialize our variables */
strcpy (buffer, "<private>");
NewList (&copy_publist);

/* Lock the public screen list */
publist = LockPubScreenList ();

/* and copy it */
for (psnode = (struct PubScreenNode *) publist->lh_Head;
     psnode->psn_Node.ln_Succ;
     psnode = (struct PubScreenNode *) psnode->psn_Node.ln_Succ)
{
    if (copy_psnode = AllocMem (sizeof (struct PubScreenNode), MEMF_CLEAR))
    {
        /* Copy the structure */
        *copy_psnode = *psnode;

        /*
         * ln_Name points to the public screen name, make your own copy
         */
        if (copy_psnode->psn_Node.ln_Name =
            AllocMem( strlen(psnode->psn_Node.ln_Name) + 1, MEMF_CLEAR))
        {
            strcpy (copy_psnode->psn_Node.ln_Name, psnode->psn_Node.ln_Name);
        }

        AddTail (&copy_publist, (struct Node *) copy_psnode);
    }
}

UnlockPubScreenList ();

psnode = (struct PubScreenNode *) copy_publist.lh_Head;
while ((copy_psnode = (struct PubScreenNode *)
        psnode->psn_Node.ln_Succ) &&
        (cs == NULL))
{
    if (psnode->psn_Screen == scr)
    {
        strcpy (buffer, psnode->psn_Node.ln_Name);
        cs = psnode->psn_Screen;
    }

    if (psnode->psn_Node.ln_Name)

```



```

    {
        FreeMem (psnode->psn_Node.ln_Name,
                 strlen (psnode->psn_Node.ln_Name) + 1);
    }
    Remove ((struct Node *) psnode);
    FreeMem (psnode, sizeof (struct PubScreenNode));
    psnode = copy_psnode;
}
}

```

With the AREXX interface it is often desirable to be able to pass information back to another application making inquiries. AppShell supports the Rexx Variables Interface (RVI) protocol for handing information back to such inquiries. The function **GetInfoFunc()** is an example of how to do such things.

```

VOID
GetInfoFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    /* ARExx function prototypes */
    extern LONG SetRexxVar (struct Message *, UBYTE *, UBYTE *, LONG);
    extern BOOL CheckRexxMsg (struct Message *);

    struct AppData *ad = (struct AppData *) ai->ai_UserData;
    UBYTE stem[128], value[140];
    struct MsgHandler *mh;
    struct RexxMsg *msg;
    struct Funcs *f;
    LONG kind;

    /* Build error string */
    ai->ai_Pri_Ret = RETURN_WARN;
    ai->ai_Sec_Ret = ERROR_CANT_RVI;
    ai->ai_TextRtn = PrepText (ai, APSH_USER_ID, ai->ai_Sec_Ret, NULL);

    /* Find out what type the current message is */
    if (GetAPSHAttr (ai, APSH_ActvMH, ai, &kind))
    {
        /* See if the current message is an ARExx message */
        if (kind == APSH_AREXX_ID)
        {
            /* Get a pointer to the current message */
            if (GetAPSHAttr (ai, APSH_ActvMessage, ai, &msg))
            {
                /* Check to see if we can set an RVI on this message. */
                if (CheckRexxMsg (msg))

```

```

{
    /* Get the destination stem name */
    sprintf (stem, "%s.", ai->ai_BaseName);
    if (f = af->af_FE)
    {
if (f->fe_Options[0])
    {
        strcpy (stem, (STRPTR) f->fe_Options[0]);
    }
    }

/* Send the application version */

    /* Build the version variable */
    sprintf (ad->ad_TmpText, "%sVERSION", stem);
    strcpy (value, ai->ai_AppVersion);

    /* Set the RVI's */
    SetRexxVar (msg, ad->ad_TmpText, value, strlen (value));

/* Send the current public screen name */

    /* Build the screen name variable */
    value[0] = 0;
    sprintf (ad->ad_TmpText, "%sSCREEN", stem);
    if (ai->ai_ScreenName)
    {
strcpy (value, ai->ai_ScreenName);
    }
    else
    {
PubScreenName (ai->ai_Screen, value);
    }

    /* Set the RVI's */
    SetRexxVar (msg, ad->ad_TmpText, value, strlen (value));

/* Send the ARExx port name */

    /* Build the ARExx port name variable */
    sprintf (ad->ad_TmpText, "%sAREXX", stem);

    /* Build the value string */
    strcpy (value, "<unnamed>");

```

```

        if (GetAPSHAttr (ai, APSH_ARExxMH, ai, &mh) && mh)
        {
strcpy (value, mh->mh_PortName);
        }

        /* Set the RVI's */
        SetRexxVar (msg, ad->ad_TmpText, value, strlen (value));

/* Send the current mode */

        /* Build the variable name */
        sprintf (ad->ad_TmpText, "%sMODE", stem);

        /* Build the value string */
        strcpy (value, "<unset>");
        if (ad->ad_Mode == 0)
strcpy (value, "Male");
        else if (ad->ad_Mode == 1)
strcpy (value, "Female");
        else if (ad->ad_Mode == 2)
strcpy (value, "Zombie");

        /* Set the RVI's */
        SetRexxVar (msg, ad->ad_TmpText, value, strlen (value));

/* Return the name of the destination stem variable */

        /* Let them know what the stem variable is named */
        strcpy (ad->ad_TmpText, stem);
        ai->ai_Pri_Ret = RETURN_OK;
        ai->ai_Sec_Ret = NULL;
        ai->ai_TextRtn = ad->ad_TmpText;
    }
    }
}
}
}
}

```

When the application wants to shut down, it gets many chances to handle any shutdown contingencies. Indeed, by using functions such as **QuitFunc()** its possible to limit or even disable application shutdown if, for example, the application is in a critical state or calculation.

VOID

QuitFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)

```

{
    /* Tell the AppShell that we're all done now. */
    ai->ai_Done = TRUE;
}

```

The word "Blurp" means nothing. It was chosen as a mnemonic for the act of pulling one's self up by his roots and moving to a new location (an apt metaphor for the behind-the-scenes actions AppShell really takes to move the application from one screen to another). This said, the purpose of the **BlurpFunc()** routine as a demonstration of screen-jumping makes sense.

While screen-jumping was a necessary feature, the time to implement it and the conditions under which it was implemented were rather difficult. For this reason, the implementation isn't yet as clean as it could be, and worse, a more viable implementation is going to require code modifications of application code. It was a case of need beating utility.

This feature is new for V37 AppShell. The way a screen-jump is done is to shut down the entire Intuition interface and dynamically change the configuration of an "anchor" window. This is done by reopening the Intuition handler inside a Deactivate/Activate pair. At that point a new screen environment can be handed into the Intuition message handler, which will cause the application to "jump" to the new screen (creating it, if necessary).

```

VOID
BlurpFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    extern struct TagItem mainenv2[];
    extern struct TagItem stags2[];

    PerfFunc (ai, DeActivateID, NULL, NULL);

    HandlerFunc (ai,
        APSH_Handler, "IDCMP",
        APSH_Command, APSH_MH_OPEN,
        APSH_WindowEnv, (ULONG) mainenv2,
        APSH_NewScreenTags, (ULONG) stags2,
        TAG_DONE);

    PerfFunc (ai, ActivateID, NULL, NULL);
}

```

Finally, the **PingFunc()** exists solely as a simple demonstration of another new V37 AppShell ability, notification of Intuition Activate/Deactivate commands. There are ActiveB/ActiveA (before and after) hooks for Activate events, and matching DeactiveB/DeactiveA hooks for Deactivate events. For more information on the new V37 additions to the Intuition message handler, please check the Intuition handler command reference in the appendices.

```

VOID PingFunc (struct Hook *h, struct AppInfo *ai, struct AppFunction *af)
{
    /* Do-nothing function */
}

```

```
DisplayBeep(NULL);  
}
```

Appendix A

Command Reference: AppShell Kernel

This section of the AppShell Kernel has an interface which is essentially the same as any of the message handlers. The primary difference is that instead of using a message handler as an interface to the user or some other aspect external to the application, the AppShell "handler" is how the application interfaces with AppShell itself. More specifically, the AppShell interface provides control over the application's own execution environment.

Handler Name:	AppShell
Purpose:	Application interface to the AppShell kernel.
Used by:	All
Maturity:	Stable

Handler Function Tags

APSH_NumArgs

This tag is used to specify the number of Shell arguments passed. This tag is used by the Clone or Tool feature and should not be used otherwise.

APSH_ArgList

This tag is used to specify the Shell arguments passed. This tag is used by the Clone or Tool feature and should not be used otherwise.

APSH.WBStartup

This tag is used to specify the Workbench arguments passed. This tag is used by the Clone or Tool feature and should not be used otherwise.

APSH_ControlPort

This tag is used to specify the SIPC control port for the application. This tag is used by the Clone or Tool feature and should not be used otherwise.

APSH_Name**APSH_Version****APSH_Copyright****APSH_Author**

Used to specify the information about the application. To be shown in the Standard About requester along with the image for ai_ProgDO.

APSH_AppMsgTitle

Used to specify the title to display in the error and message EasyRequest.

APSH_DefText**APSH_AppCatalog****APSH_AppDefLang**

These define the application's default text table and locale information. APSH_DefText gets the pointer to the default text array generated by CatComp. APSH_AppCatalog takes a pointer to a charstring which gives the catalog name (for example, "skeleton.catalog"). Finally, APSH_DefLang takes a pointer to a charstring which gives the language used for the application's default strings ("english", etc.).

APSH_AppInit

Function ID to dispatch after the message handlers have been initialized, and before entering the event processing stage.

APSH_AppExit

Function ID to dispatch after ai_Done has been set to TRUE, and before running shut down on the message handlers.

APSH_SIG_C

Function ID to dispatch when the corresponding signal is sent to the application. Defaults to QuitID.

APSH_SIG_D**APSH_SIG_E****APSH_SIG_F**

Function ID to dispatch when the corresponding signal is sent to the application.

APSH_BaseName

Base name assigned to the application. The public screen name, ARexx message port and other public names are derived from this.

APSH_Template

ReadArgs template to use when the application is started from the Shell.

APSH_NumOpts

Number of options specified in APSh.Template.

APSH_UserData

Pointer to preallocated user data for the application. ai_UserData will be set to this value.

APSH_UserDataSize

If this tag is specified, then the AppShell will allocate this much memory off of the ai_UserData field at initialization time, and free it at shutdown.

Standard Functions**Command: Alias**

Template: NAME, COMMAND/F

Link a new command name to an old command name.

Command: Disable

Template: /M

Disable a function or a set of functions. If there is a GUI item, then it will disable that item also. If the function is bound to a menu item, then it disables that menu item. Currently only disables functions and gadgets. Supports multiple names and pattern matching.

Command: Enable

Template: /M

Enable a function or a set of functions. Currently only enables functions and gadgets. Supports multiple names and pattern matching.

Command: Fault

Template: /N

Return the text string assigned to an error number.

Command: Stub

Template: ,

Function which does nothing.

Command: Version

Template: APPSHELL/S

Display the version string of the application. When the APPSHELL switch is set, then display the version of the AppShell.

Appendix B

Command Reference: UI Handlers

The following is an overview of each of the message handlers that are provided in the AppShell shared library. Information is given on:

- Short name of the message handler.
- List of tags that the message handler can translate.
- Standard Functions that the message handler implements.
- Extended low-level functions that the message handler implements.
- The preference files where the message handler gets user settings.
- A short example on how an application can utilize the message handler.

B.1 AREXX

The ARexx message handler provides a standard scripting language, as well as string oriented interprocess communications. Currently the ARexx message handler only implements commands.

Handler Name:	AREXX
Purpose:	Provides application with an AREXX interface
Used by:	App
Maturity:	Stable

Handler Function Tags

APSH_Extens

ARexx macro file name extension. Defaults to .REXX.

APSH_ARexxError

Function ID to dispatch when there is an ARexx command error.

APSH_ARexxOK

Function ID to dispatch when an ARexx command succeeds.

APSH_Port

Base name for the applications public ARexx port. Defaults to the application's base name.

APSH_Status

Recognizes:

- **APSHP_INACTIVE**
When set, leaves the ARexx message handler inactive. Causes the message handler to reply to each incoming message with the return value set to RETURN_FAIL. When not set, causes ARexx to become immediately available.
- **APSHP_SINGLE**
When set, uses the base port name as is. When not set, will append .# to the base port name, where # is incremented until it finds a unused name.

Extended Low-Level Message Handler Functions

AH_SENDCMD

Sends the passed command (APSH_CmdString) to ARexx.

Standard Functions

Command: **RX**

Template: **COMMAND/f**

Execute an ARexx command. Only to be used when there is a name conflict between an ARexx command name and a function short name.

Command: **Why**

Template: **,**

Return more information on the last error. The information is placed in the ai.TextRtn field, and could be text or a number sprintf'ed into text. In an ARexx script, the primary return value should indicate failure level. This command allows us to get at the real reason a command may have failed. This function is currently not implemented.

Preferences

None.

Example

```
/* ARexx user interface environment specification array */
struct TagItem Handle_AREXX[] =
{
    {APSH_Extens, (ULONG)"proj"},
    {APSH_Rating, APSH_OPTIONAL},
    {TAG_DONE,}
};

/* Tell about our application */
struct TagItem Our_App[] =
{
    {APSH_AddARexx_UI, (ULONG)Handle_AREXX},
    {TAG_DONE,}
};

/* ... other tags go here... */

struct Library *AppShellBase;
extern struct WBStartup *WBenchMsg;

/* Main processing loop */
VOID main (int argc, char **argv)
{
    /* open the AppShell library */
```

```
        if (AppShellBase = OpenLibrary ("appshell.library", 36))
        {
            /* Main AppShell entry point */
            HandleApp (argc, argv, WBenchMsg, Our_App);

            /* close the AppShell library */
            CloseLibrary (AppShellBase);
        }
    }
```

B.2 Command Shell

The Command Shell Message Handler provides a command shell for the more advanced user. It allows the user to directly dispatch any of the functions in the function table. It also gives direct access to the power of ARexx without always having to specify the ADDRESS or macro file name extension.

Handler Name:	DOS
Purpose:	Provides a CLI-like command shell.
Used by:	App
Maturity:	Stable

Handler Function Tags

APSH_CloseMsg

Text ID of the message to display in the Shell before attempting to close it. Defaults to "Waiting for macro return".

APSH_CMDWindow

Text ID of the initial Shell window specification. Defaults to "CON:0/150/600/50/Command Shell/CLOSE/AUTO".

APSH_Prompt

Text ID of the initial Shell prompt. Defaults to "Cmd>".

APSH_CMDTitle

Used to specify the text ID of title for the Command Shell. Defaults to "{basename} Command Shell".

APSH_CMDParent

Used to specify the name of the parent window of the Command Shell. Should be the main project window of the application. Using this tag helps the AppShell keep the Command Shell near the application when the user is using a virtual screen. Not fully implemented yet.

APSH_Status

Recognizes:

- **APSH_INACTIVE**

If set, keep the Command Shell window closed until specifically requested to open. If not set, then open the Command Shell window at initialization time.

Extended Low-Level Message Handler Functions

None.

Standard Functions

Command: **CmdShell**

Template: **OPEN/S, CLOSE/S, TITLE/K, SNAPSHOT/S, ACTIVATE/S, FRONT/S, BACK/S**

CmdShell allows the user to manipulate the Command Shell.

OPEN

Opens the Command Shell window, respecting the current snapshot.

CLOSE

Closes the Command Shell window.

TITLE

Set the Command Shell's window title bar.

SNAPSHOT

Save the current rectangle of the window to a file.

ACTIVATE

Activate the Command Shell window.

FRONT

Bring the Command Shell window to the front of other windows.

BACK

Send the Command Shell window to the back of the other windows.

Preferences

commandshell.win

Window size and placement preferences.

Example

```
/* Command Shell user interface environment specification array */
struct TagItem Handle_DOS[] =
{
    {APSH_Status, APShP_INACTIVE},
    {APSH_Rating, APSh_OPTIONAL},
}
```

```

        {TAG_DONE,}
    };

    /* This is the main tag list used to describe our
     * application's multiple user interfaces. */
    struct TagItem Our_App[] =
    {
        {APSH_AddCmdShell_UI, (ULONG)Handle_DOS},

/* ... other tags go here ... */

        {TAG_DONE,}
    };

    struct Library *AppShellBase;
    extern struct WBStartup *WBenchMsg;

    /* Main processing loop */
    VOID main (int argc, char **argv)
    {
        /* open the AppShell library */
        if (AppShellBase = OpenLibrary ("appshell.library", 36))
        {
            /* Main AppShell entry point */
            HandleApp (argc, argv, WBenchMsg, Our_App);

            /* close the AppShell library */
            CloseLibrary (AppShellBase);
        }
    }

```

B.3 Intuition (IDCMP)

The IDCMP message handler provides the application with a Graphical User Interface (GUI) through the use of Intuition, GadTools and AppObjects.

Handler Name:	IDCMP
Purpose:	Provides application with GUI.
Used by:	App
Maturity:	Recently changes (major)

The handler maintains such things as:

- **Public or private screens** - Manages the DrawInfo and VisualInfo for the application screen. Also tracks the public screen information whenever appropriate.
- **Multiple windows** - Manages an unlimited number of application windows.
- **Translating Workbench preferences** - Translate user-interface object description, based on user preferences, into GadTools and Intuition objects. For instance, it will scale the objects according to the font; or remap pens according to the color palette. Will also load the normal and wait pointers.
- **Snapshotting window preferences** - Manages saving/restoring user preferences for individual windows. Includes such things as placement, and size.
- **GadTool gadgets** - Converts object descriptions into GadTools structures and tags.
- **boopsi objects** - Supports boopsi objects.
- **Images, borders and text** - Object management of images, text and common border types.

To define GUI items such as windows, buttons, sliders and scrolling lists, the application uses an array of Object structures. Here is what the Object structure looks like (defined in *libraries/appshell.h*):

```
struct Object
{
    struct Object *o_NextObject;    /* next object in array */
    WORD o_Group;                  /* Object group */
    WORD o_Priority;                /* Inclusion priority of object */
    ULONG o_Type;                  /* type */
    ULONG o_ObjectID;              /* ID */
    ULONG o_Flags;                 /* see defines below */
    UWORD o_Key;                   /* hotkey */
    STRPTR o_Name;                 /* name */
    ULONG o_LabelID;               /* label index into text catalog */
    struct IBox o_Outer;           /* size w/label */
    struct TagItem *o_Tags;        /* tags for object */
    APTR o_UserData;               /* user data for object */
};
```

o_NextObject

Pointer to the next object in the array.

o_Group

The layout group that the object belongs in.

o_Priority

The priority assigned to the object or group. Zero indicates high priority.

o_Type

Indicates type of object. Valid object types are shown in the next section.

o_ObjectID

Function ID of the command to call when an event occurs for this object.

o_Flags

Flags which define the behavior of the object.

- **APSH.OBJF_CLOSEWINDOW**
Indicates that this object will cause the window to close.
- **APSH.OBJF_ACTIVATE**
Indicate that this is the default object to activate when the window becomes active.

o.Key

Keystroke binding for this object. Whenever the key is pressed, the corresponding function for the gadget is executed. On the downpress of the key, then EG_DOWNPRESS is executed, repeat calls EG_HOLD, and release calls EG_RELEASE. If the label for the object contains an '_' sign in it, then the key following the sign will become the keystroke binding for the object. This for keystroke internationalization also.

o.Name

All objects are placed into a list for the window that they belong in. o.Name is the name that is used for the list entry (node) for this object. This name should be unique to this window.

o.LabelID

A numeric ID assigned to the visual text label for this object. This corresponds with the string ID given in the locale catalog and default string array.

o.Outer

The IBox which describes the placement and size of an entire object, including the visual label, if pertinent. A negative number for any value indicates that that value is relative. Currently the GadTools objects don't support relativity.

- **Left**
When positive indicates LEFT, when negative indicates GRELRIGHT.
- **Top**
When positive indicates TOP, when negative indicates GRELBOTTOM.
- **Width**
When positive indicates WIDTH, when negative indicates GRELWIDTH.
- **Height**
When positive indicates HEIGHT, when negative indicates GRELHEIGHT.

o.Tags

Additional parameters for defining the object. Following are the generic tags used by the gadget-like objects.

- **APSH.GTFlags**
GadTools NewGadget flags to use for this object.
- **APSH.ObjDown**
Function ID to dispatch on the downpress of an Intuition gadget.
- **APSH.ObjHold**
Function ID to dispatch when an Intuition gadget is being held down.
- **APSH.ObjRelease**
Function ID to dispatch on the release of an Intuition gadget.

- **APSH.ObjDbClick**
Function ID to dispatch when an Intuition gadget has been double-clicked.
- **APSH.ObjAbort**
Function ID to dispatch when the right mouse button has been pressed while an Intuition gadget is active.
- **APSH.ObjAltHit**
Function ID to dispatch when an Intuition gadget is selected while holding either ALT key.
- **APSH.ObjShiftHit**
Function ID to dispatch when an Intuition gadget is selected while holding either SHIFT key.
- **APSH.ObjExtraRelease**
Used to specify the function for the button added to OBJ_DirString or OBJ_DirNumeric.
- **APSH.ObjCreate**
Function ID to dispatch after the object has been created.
- **APSH.ObjDelete**
Function ID to dispatch before the object may be deleted.
- **APSH.ObjUpdate**
Function ID to dispatch when an IDCMP_IDCMPUPDATE event occurs for the gadget. The AppShell searches for the object that is associated with the GA_ID attribute when the event occurs.
- **APSH.ObjData**
Used to provide the additional data for the object. Used to specify the image for OBJ_Display, OBJ_Select, OBJ_Dropbox, OBJ_GImage, and OBJ_Image.
- **APSH.ObjAltData**
Used to specify the alternate image for an object.

o UserData

Pointer available to the application for binding data to a particular object. Note that the gadget's UserData field is currently used by the AppShell, and application user data is assigned to the ObjectNode's UserData field.

Following are the valid object types (defined in *libraries/appshell.h*). They are broken into type; GadTool, boopsi, image and group.

The tags that each object understands are listed below the object type. The tags are prefixed with a character that indicates when the tag can be specified. An 'I' indicates that it can be set at initialization time. A 'S' indicates that it can be manipulated with the SetGadgetAttrs() or GT_SetGadgetAttrs(). A 'G' indicates that it can be obtained with GetAttr(). A 'N' indicates that the attribute is sent out to the APSH_ObjUpdate function.

The following objects are GadTool gadgets. They accept the GadTool tags defined for the particular gadget type. Attributes are set with the GT_SetGadgetAtts() call.

OBJ_Generic

OBJ_Checkbox

Check box for boolean values. o_ObjectID is called on downpress. Height is font height.

IS GTCB_Checked

OBJ_Integer

String gadget for numeric entry. o_ObjectID is called on downpress. Height is font height plus six.

I GTIN_MaxChars

IS GTIN_Number

OBJ_Listview

Scrolling list gadget. o_ObjectID is called on downpress. Minimum width of fifty pixels, minimum height of thirty pixels.

I Use the APSH_ShowSelected tag to specify the name of an OBJ_Integer or OBJ_String gadget to attach to the bottom of the list view.

I GTLV_ReadOnly

IS GTLV_Top

IS GTLV_Labels

IS GTLV_Selected

OBJ_MX

Mutual exclusion gadget. o_ObjectID is called on downpress.

I GTMX_Spacing

IS GTMX_Labels

IS GTMX_Active

OBJ_Number

Numeric display box. Height is font height plus six.

I GTNM_Border

IS GTNM_Number

OBJ_Cycle

Cycle gadget. o.ObjectID is called on release. Height is font height plus six. Minimum height of sixteen.

- IS GTCY_Labels
- IS GTCY_Active

OBJ_Palette

Color palette gadget. o.ObjectID is called on downpress.

- IS GTPA_Color

OBJ_Slider

Slider gadget.

- IS GTSL_Min
- IS GTSL_Max
- IS GTSL_Level
- IS GTSL_MaxLevelLen
- IS GTSL_LevelFormat
- IS GTSL_LevelPlace
- IS GTSL_DisFunc

OBJ_String

String gadget for text entry. o.ObjectID is called on RETURN. Height is font height plus six.

- I GTST_MaxChars
- IS GTST_String

OBJ_Text

Text display box. Height is font height plus six.

- I GTTX_Border
- IS GTTX_Text
- IS GTTX_CopyText

The following gadgets are boopsi objects. Attributes are set with the Intuition **SetGadgetAttrs()** function. Attributes may also be read by using the **GetAttrs()** function.

OBJ_Button

Action gadget. o.ObjectID is called on release. Height is font height plus six. Label is centered within object.

- I Uses o_LabelID for the text to display.
- SG GA_TEXT

OBJ_Scroller

Scroll gadget. o_ObjectID is called on release, and for changes.

- ISGN GTSC.Top - Top line of the view.
- ISGN GTSC.Total - Total number of lines.
- ISGN GTSC.Visible - Number of lines in view.
- ISGN GTSC.Overlap - Amount of overlap when scrolling pages.
 - S CGTA.Increment - Increment top by one.
 - S CGTA.Decrement - Decrement top by one.

OBJ_Display

Freeform display box for text or images.

- I Uses o_LabelID for the text to display.
- SG GA.TEXT
- I Specify the APSH.ObjData tag in order to use an Intuition image.
- SG GA.LABELIMAGE

OBJ_Select

Freeform action gadget for text or images.

- I Uses o_LabelID for the text to display.
- SG GA.TEXT
- I Specify the APSH.ObjData tag to use an Intuition image instead.
- SG GA.LABELIMAGE
- I In order to use a boopsi image, use the APSH_GA_LabelImage tag to provide the name of an OBJ_boopsi object.
- SG GA.LABELIMAGE

OBJ_Dropbox

AppWindow icon drop box. Specify the APSH.ObjData to specify an image to display. Size is ninety pixels wide by forty-four pixels high.

- I Use the APSH.ObjData tag to specify the initial image to display.
- SG GA.LABELIMAGE
- I Set the APSH.OBJF_DRAGGABLE bit of o_Flags to indicate that the image is draggable.

OBJ_GImage

Iconic action gadget (image, but no 3D border).

- I Use the APSh.ObjData tag to specify the initial image to display.
- SG GA_LABELIMAGE
- I Set the APSh.OBJF_DRAGGABLE bit of o_Flags to indicate that the image is draggable.

OBJ_MultiText

Multiple line wrapping text gadget. Minimum width of fifty pixels, minimum height of thirty pixels.

- I STRINGA_MaxChars - Buffer size, in characters, to allocate.
- I STRINGA_Buffer - Application provided buffer.
- I STRINGA_UndoBuffer - Application provided undo buffer.
- ISGN STRINGA_TextVal - Displayed text.
- IS STRINGA_WorkBuffer
- IS STRINGA_BufferPos
- IS STRINGA_DisPos
- IS STRINGA_AltKeyMap
- IS STRINGA_Font
- IS STRINGA_Pens
- IS STRINGA_ActivePens
- IS STRINGA_EditLook
- IS CGTA_HighPens
- I CGTA_DisplayOnly
- ISG CGTA_Top
- G CGTA_Visible
- G CGTA_Total

OBJ_DirString

Text gadget with button for directory listing. Height is font height plus six.

OBJ_DirNumeric

Numeric gadget with button for directory listing. Height is font height plus six.

OBJ_boopsi

boopsi gadget or image.

I Use the APSH.GA_Image tag to specify the name of the OBJ_boopsi object to use as the GA_Image object.

S GA_Image

I Use the APSH.GA_SelectRender tag to specify the name of the OBJ_boopsi object to use as the GA_SelectRender object.

S GA_SelectRender

OBJ_View

Multi-column, multi-select view.

I AOLV_Borderless - Place a border around the view?

I AOLV_Freedom - Horizontal and/or Vertical.

I AOLV_ReadOnly - List read-only?

I AOLV_MultiSelect - List support multi-selection?

I AOLV_ControlHeight - Height of the control panel.

I AOLV_LabelHeight - Height of the column label bar.

IS AOLV_UnitHeight - Height of each row. Use 0 to use the screen font height plus one.

IS AOLV_UnitWidth - Width of each horizontal unit.

IS AOLV_List - List to display.

IS GTLV_Labels - Same as AOLV_List.

ISGN GTLV_Selected - Currently selected vertical row.

ISGN AOLV_TopVert - Top vertical line.

ISGN AOLV_VisibleVert - Number of visible vertical lines.

ISGN AOLV_TotalVert - Number of total vertical lines.

ISGN AOLV_TopHoriz - Top horizontal line.

ISGN AOLV_VisibleHoriz - Number of visible horizontal lines.

ISGN AOLV_TotalHoriz - Number of total horizontal lines.

GN AOLV_View - View rectangle.

OBJ_MListView

Multi-column, multi-select scrolling list view. Same tags as the OBJ_View object.

The following objects are for images.

OBJ_Image

Display an image.

I Use the APSH.ObjData tag to specify the initial image to display.

SG GA_LABELIMAGE

OBJ.Column

Column image, that can only be used with OBJ_View and OBJ_MListView

- I AOLV_ColumnWidth - Percent of total width. This is stored in image->ImageData.
- I AOLV_Title - Title for the column.
- I AOLV_Justification - Left or right justification.
- I AOLV_FieldOffset - Byte offset of field within structure. If zero, then use the node->In_Name field.
- I AOLV_FieldType - Type of field, such as text, text pointer, long, etc.

The following objects are different types of borders.

OBJ_BevelIn

Pushed in 3D border.

OBJ_BevelOut

Pushed out 3D border.

OBJ_DblBevelIn

Pushed in 3D embossed border.

OBJ_DblBevelOut

Pushed out 3D embossed border.

- I Use the APSH_ObjData tag to specify the initial image to display.
- SG GA_LABELIMAGE
- I Set the APSH_OBJF_DRAGGABLE bit of o.Flags to indicate that the image is draggable.

The following objects are basically different types of groups.

OBJ_Window

Window

OBJ_Group

Freeform layout group.

OBJ_VGroup

Vertical layout group.

OBJ_HGroup

Horizontal layout group.

OBJ_MGroup

Mutual exclusion group. Members can only be OBJ_Button or OBJ_Select.

OBJ_VFill

Vertical fill area.

OBJ_HFill

Horizontal fill area.

Handler Function Tags**APSH_TextAttr**

Text attribute to use.

APSH_NewScreen

NewScreen structure.

APSH_NewScreenTags

Tags to use when opening the screen.

APSH_Palette

Color palette to assign to the screen.

APSH_DefWinFlags

Default window flags.

APSH_WindowEnv

Used to describe a window environment.

APSH_NameTag

Name to assign to this window.

APSH_GTMenu

GadTools-style NewMenu array.

APSH_HotKeys

Keystroke array used to specify the function binding for any particular key.

APSH_NewWindow

NewWindow structure.

APSH.NewWindowTags

Tags to use when opening the window.

APSH.Objects

Array of objects in the Graphical User Interface. Translated into GadTools or Intuition objects, according to user preferences.

Extended Low-Level Message Handler Functions**IH.DELWIN**

This deletes the internal status information associated with a window. It accepts the following tag:

APSH.WindowEnv

This is the WindowEnv taglist for the window to be deleted.

Standard Functions**Command: Activate**

Template: ,

Restore the graphical user interface of an application to the state that it was in before receiving a DEACTIVATE command.

Command: Deactivate

Template: ,

Shutdown the entire graphical user interface of an application.

Command: Keyboard

Template: KEY, WINDOW/K, GLOBAL/S, HOTKEY/S, PROMPT/S, CMD/F

Allows the user to bind a function to a key.

KEY

Specifies which keystroke to edit.

WINDOW

Specify the name of the window that the keystroke is active in. The window that the keystroke will be used in. Defaults to MAIN.

GLOBAL

Indicate that the keystroke applies to all windows for the application.

HOTKEY

Indicate that the keystroke is system-wide, regardless of active window.

PROMPT

Bring up a window that allows a GUI for selection of keystrokes and functions.

CMD

The command to assign to the keystroke.

Command: **Window**

Template: NAME/M, TITLE/K, OPEN/S, CLOSE/S, SNAPSHOT/S, ACTIVATE/S,
MIN/S, MAX/S, FRONT/S, BACK/S, ZOOM/S, UNZOOM/S, LOCK/S, UNLOCK/S
Allows the user to manipulate a window.

NAME

Specify the window to manipulate. Defaults to MAIN.

OPEN

Open the window.

CLOSE

Close the window.

TITLE

Specify the window title.

SNAPSHOT

Save the current window rectangle information to a file.

ACTIVATE

Activate the window.

MIN

Size the window to its minimum rectangle.

MAX

Size the window to its maximum rectangle.

FRONT

Send the window to the front of other windows.

BACK

Send the window behind all others.

ZOOM

Change the window box to its zoomed position.

UNZOOM

Change the window box to its unzoom position.

LOCK

Lock all input to the window.

UNLOCK

Unlock a previous LOCK.

Command: **WindowToBack**

Template: NAME

Sends the named window to back. Defaults to MAIN.

Command: **WindowToFront**

Template: NAME

Brings the named window to front. Defaults to MAIN.

Preferences

The AppShell will monitor user preference files. [I] indicates implemented, [NI] indicates not implemented.

palette.prefs [I]

Colors to use in a custom or public screen. Automatically provides a pen spec.

pointer.prefs [I]

Normal pointer.

busypointer.prefs [I]

Busy pointer.

{name}pointer.prefs [NI]

Other pointers, such as mode (fill, text, etc) indicators, whenever appropriate.

screenmode.prefs [I]

Screen mode/type.

screenfont.prefs [NI]

Screen font, used for menus and window title text.

sysfont.prefs [NI]

Font to use inside the windows.

win.pat [NI]

Backfill pattern for windows.

wb.pat [NI]

Backfill pattern for background window or screen.

printer.prefs [NI]

Printer preferences.

printergfx.prefs [NI]

Graphic printer preferences.

{window title}.win [I]

Window size and placement preferences.

Example

Due to the extensive size and file requirements to demonstrate the structure of IDCMP handler code, please refer to the "Skeleton" example cited earlier in this document.

B.4 Notification

The Notification message handler provides a way to watch a file and execute a function whenever that file is changed. This message handler is used to provide preferences for an AppShell application, and is therefore added to the application's message handler list automatically.

Handler Name:	NOTIFY
Purpose:	Application interface to file notification.
Used by:	All
Maturity:	Stable

Handler Function Tags

APSH_NameTag

Complete file name of file to watch.

APSH_CmdID

Function to execute when file is modified. Examine APSH_NameTag to get the name of the file modified.

Extended Low-Level Message Handler Functions

None.

Standard Functions

None.

Preferences

None.

Example

NOTE: Notification is started with the APSH_MH_OPEN command and can be stopped with the APSH_MH_CLOSE command. The Notification message handler will automatically terminate any outstanding notification requests when it does a shutdown.

```
/* Start notification on a file */
VOID
WatchClipFile (struct Hook *h,
               struct AppInfo *ai,
               struct AppFunction *af)
```

```

{
    /* Start the notification */
    HandlerFunc (ai,
        APSH_Handler, "NOTIFY",
        APSH_Command, APSH_MH_OPEN,
        APSH_NameTag, "ram:temp",
        APSH_CmdID, ClipFileChangeID,
        TAG_DONE);
}

/* Function that gets called whenever file is changed */
VOID
ClipFileChange (struct Hook *h,
                 struct AppInfo *ai,
                 struct AppFunction *af)
{
    struct TagItem *attrs = af->af_FE;
    STRPTR name;

    /* Get the name of the file that changed */
    name = (STRPTR) GetTagData (APSH_NameTag, NULL, attrs);
}

```


B.5 Simple Inter-Process Communication (SIPC)

The Simple Interprocess Communications Message Handler (SIPC) provides fast and simple communications between applications and tools.

The tool message handler uses the SIPC to tell tools and applications to shut down. The HyperText library uses SIPC to talk to an application that supports it.

Handler Name:	SIPC
Purpose:	Direct interface to the application command bus.
Used by:	App, Tool, Clone
Maturity:	Stable

Handler Function Tags

APSH_Port

Used to provide a name for the SIPC port. Defaults to basename plus .SIPC

APSH_Status

Recognizes:

- **APSHP_INACTIVE**
When set, leaves the SIPC message handler inactive. Causes the message handler to reply to each incoming message with the return value set to RETURN_FAIL. When not set, causes SIPC to become immediately available.
- **APSHP_SINGLE**
When set, uses the base port name as is. When not set, will append .# to the base port name, where # is incremented until it finds a unused name.

Extended Low-Level Message Handler Functions

AH_SENDCMD

Sends the a command to the designated SIPC port. Understands the following tags:

- **APSH_NameTag**
Specify the name of the SIPC port to send the command to.
- **APSH_PortAddr**
Specify the address of the SIPC port to send the command to.
- **APSH_CmdID**
Function for the destination application to execute.

- **APSH.CmdData**
Data to pass to the destination application.
- **APSH.CmdDataLength**
Length of the data block to pass. If zero or not present, then APSH.CmdData is assumed to be a TagItem array.
- **APSH.CmdString**
Specify a command string to send to the destination application.

Standard Functions

None.

Preferences

None.

B.6 Tool

The Tool Message Handler provides the application with the capability to run asynchronous processes. If an application allows multiple projects, the tool message handler offers a way for the application to start a new process for each project. It also allows tools relevant to the application to be run asynchronously. Such tools could be a color cyler, animator or real time display.

Handler Name:	TOOL
Purpose:	Provides application with ability to launch other programs, AppShell-based or not.
Used by:	App
Maturity:	Stable

Handler Function Tags

APSH.Tool

Name to give to the process that is spawned. Required.

APSH.ToolData

Data to pass to the new process. Passes a pointer to the AppInfo structure by default. If you are starting a new AppShell application, then a tag list should be passed.

APSH.ToolAddr

Pointer to the function to spawn as a new process. Defaults to HandleAppAsync.

APSH.ToolStack

Stack size to allocate for the new process. Defaults to 4096 bytes.

APSH.ToolPri

Priority to run the new process at. Defaults to zero.

Extended Low-Level Message Handler Functions

None.

Standard Functions

None.

Preferences

None.

Appendix C

AppBuilder Design Notes

The purpose of this document is to describe the following:

- Overview of AppBuilder's goals
- Overview of AppBuilder's features

C.1 Overview of AppBuilder's Goals

In the past, Amiga User Interface Building Tools have focused solely on defining an application's Graphic User Interface (GUI). INOVAtronics' product Power Windows, released in 1987, aided in creating GUI's using all major Intuition constructs existing prior to release 2.0 of the Amiga operating system. For version 2.0 of the Amiga OS, Commodore-Amiga's Toolmaker, released in mid-1992, aided in creating GUI's using the constructs available in the Gadtools library.

However, for Amiga applications, the GUI is just one of the user interfaces which an application should provide. Application users are beginning to expect other methods of interfacing with their software.

In 1991 Commodore-Amiga, Inc. released the Amiga User Interface Style Guide (AUISG) [Com91d] describing suggested standards for Amiga Application User Interfaces. It identified three forms of Amiga user interfaces:

- A graphic user interface
- A scripting language that can handle inter-process communication (ARexx)
- A command line interface

In addition, because of the Amiga's world market, the AUISG identified a factor that was not addressed by previous GUI construction tools: internationalization.

These two factors, the Amiga's multiple user interfaces and the need for localization, required a new approach to be taken by Amiga User Interface Building Tools.

AppBuilder is designed around a model of applications that was introduced in AppShell. This model divides an application into four main components:

- **The User Interfaces** – how the user views and controls the application.
- **A Localizable Text Table** – the text which will appear in the user interface, but which can be translated to different languages.
- **Application Functions** – the operations that are unique to the application (e.g. recalculating cells in a spreadsheet.)
- **Standard Functions** – the operations necessary in all applications, such as proper event management, ARexx command dispatching, argument parsing, and locale management. In other words, a framework of functions which manage the first three components of an application (e.g. the AppShell Library).

AppBuilder provides a graphic user interface for defining (and establishing the relationships between) the first three of these components. The fourth component is provided by the source code which AppBuilder generates. Because of AppBuilder's programmable source code generator (described below), a choice of frameworks is possible. AppBuilder will initially provide a choice between the AppShell framework and a framework which uses Gadtools. The generated source code can be compiled (or assembled) and linked with the Application Functions to result in a usable application.

C.2 Overview of AppBuilder's features

The rest of this document will describe AppBuilder's three main components:

- The Interface Design Editors
- The Object Librarian
- The Programmable Source Code Generator

C.2.1 The Interface Design Editors

The graphic user interface for AppBuilder consists of a collection of separate requesters, each providing an interface for editing a component of a project's definition. AppBuilder can load multiple projects, each containing the complete user interface definition for a single application. AppBuilder's main requester is the Project Editor. This displays the name of each project in a Listview, and indicates which one is the current project. Only one project is active at a time.

With few exceptions, all requesters in AppBuilder are non-modal. This means that one requester does not block the input to another. Users can open and close the requesters as needed, leaving as many open as desirable. The information can be entered in any order, and at any time; users may switch from window to window at will. In addition, any changes made in a requester will also update any other requesters which depend on that information.

Six of AppBuilder's editors are concerned with components of the Application other than the GUI. The description of each editor is not necessarily complete, but is intended to indicate the nature of the information which it manages.

1. *The Base Description Editor*: This editor is used to specify general information about the project, such as its author, version, and copyright. The source code name of a hook handling function can also be specified here. In addition, this requester is used to bind Application Functions to initialization and exiting events. Functions may be bound to signal events as well.
2. *The System Libraries Editor*: This editor is used to specify information about the system libraries which the project's Application Functions may need.
3. *The Text List Editor*: This editor is used to define the project's Locale Text List. Each entry in the list contains a string, an ID number and, optionally, a source code label. The source code label can be used by the project's Application Function, and is defined as the Text ID value.
4. *The Function List Editor*: This editor is used to define the interface to each of the project's Application Functions. To define a Function, its source code name must be provided, and a short name must be entered. The short name is used by any user interface which sends text messages to the application (e.g. ARexx). In addition, a DOS argument template can be specified.
5. *The User Interface Settings Editor*: This editor is used to specify which user interfaces the application will support, and to specify any general attributes of each interface. AppBuilder supports four interfaces (to work with the AppShell framework): Intuition, ARexx, CommandShell, and AppShell's SIPC interface. Others can be defined using this editor.

AppBuilder provides a collection of Editors for defining the application's GUI. AppBuilder was designed according to the WYSIWYG philosophy. With few exceptions, every change made to the definition of the project's GUI will result in immediate feedback through changes in the visible GUI. There are six general categories of GUI components.

1. *The Screen Editor*: This editor is used to specify the type of screen the application will require. An application can open on a public screen, create a custom screen, or create a public screen. A default palette can also be specified.
2. *The Window Editors*: Windows are created and defined by two separate editors. One editor manages the list of Windows, specifying only a Window Name to be used by the source code generator, and the Window Title. The other editor is used to define the Window Attributes, such as its initial coordinates, zoom size, minimum and maximum sizes, and flags. This editor is also used to specify the Application Functions which are bound to Window events. An actual window is created on AppBuilder's screen, providing a representation of the Application's Window.
3. *The Gadget Editors*: These editors provide a means of adding, moving, and defining the gadgets for a window. There are three requesters involved:

- *The Gadget Type requester* – provides a list of supported gadget types, and displays an image of the current selection. This image can be clicked on and dragged to the Application's Window, then dropped where it should be added.
 - *The Gadget Editor* – displays a list of all the Gadgets that are attached to the current window. General information such as the Gadget's label and coordinates are edited from this requester.
 - *The Gadget Attributes Editor* – this requester is used to specify the type specific gadget attributes.
 - *The Application's Window* – this window can be used to select and move the gadgets in the window. The currently selected gadget is indicated by a gadget cursor which can be moved or resized.
4. *Menu Editor*: This editor provides a "drag and drop" interface for designing Amiga Menu strips. Adding an element (such as a TITLE, an ITEM, or a SUBITEM) is as easy as dragging an image to a menu display area, and dropping it where it belongs. Elements can be selected within this area and dragged to new locations. The requester provides options for defining each element.
 5. *Image Editors*: Some gadgets support images. Two editors are provided to manage the images used in an application. The first handles the a list of images, and displays the currently selected image. The second is a simple paint editor which can be used to create an image or to make changes to a previously created image. As with other editors, changes made to the images cause immediate visual updates to the interface being designed.
 6. *Keystroke Editor*: This requester is used to specify a list of hot keys for the application. Each hot key can be bound to an Application Function.

C.2.2 The Object Librarian

The purpose of the Object Librarian (OL) is to allow interfaces to be built from previously defined components. This section gives a brief overview of the types of components which the OL can manage.

- Text Lists
- Function Lists
- Images
- Gadgets
- Menus
- Windows

Each of these components can be copied from a project or copied to a project. Many of these components are themselves made up of subcomponents. For example, a Window consists of a number of gadgets, each of which may need a text label from the Text List or may require certain Application Functions to be specified in the Function List. When a Window is copied from the project, all of its subcomponents are copied as well. Likewise, when a Window is copied to a project, all of these components will also be copied to the project.

Conflicts may arise when the components that are copied from the OL already exist in the project. Through preference settings, AppBuilder provides several options for handling such conflicts.

C.2.3 The Programmable Source Code Generator

AppBuilder generates source code based on the instructions located in a user selectable Template file. A Template file contains a script written in the AppBuilder Code Description (ABCD) Language, which will be discussed below.

Four Template files are initially included with AppBuilder:

- Lattice C using AppShell
- Lattice C using Gadtools
- Metacomco Assembly using Appshell
- Metacomco Assembly using Gadtools

Benefits of ABCD

The existence of ABCD results in a number of benefits for users of AppBuilder.

First, it is possible to create Templates which generate code for other languages or for other frameworks. Such additional Templates could be placed in the public domain, or could be included by third party developers with their compilers or with their framework libraries.

Second, with new compiler technology or new language standards, certain constructs in source code files which were once considered proper may now be considered bugs. Any outdated constructs, or any other discovered bugs, found in the generated source code can be corrected by changing the Template file which produced the code. Such a change can be made by users who can't afford to wait for an upgrade to the product.

And third, the templates can be customized to conform to a developers specific programming style and needs. Any tool that requires users to manually modify – or "tweak" – the generated source code to make it usable discourages such use. This point presents the one standard we suggest for all template files:

The generated source code should not require "tweaking".

AppBuilder should be usable not just during the initial design of an application's interface, but also during any changes to that design. This goal is made difficult the moment a modification is made to the generated source code to make an application work. Once such a modification is required, it must be made again each time AppBuilder regenerates the source code. Not only can this be highly inconvenient, but it can also be highly error prone.

With ABCD, necessary modifications should be made to the *Template file* rather than to the generated source code. When this is done, all generated source code will automatically include the needed changes.

ABCD Language Functional Features

A full description of AppBuilder's Code Description Language is beyond the scope of this paper. Very briefly, ABCD is a MACRO expansion language with the following features:

- Access to all data entered through AppBuilder's Requesters.
- Multiple output files.
- User definable MACROs with parameter passing.
- User Variables.
- Comparison Operations and Case Operations.
- Column positioning with auto Tab control.
- Error Message Requesters.
- Preferences defined variables.

C.3 Conclusion

In conclusion, AppBuilder is an integrated, full featured, Amiga User Interface Building Tool. Using its rich set of GUI requesters, and the Source Code Generation Templates it provides, applications authors will have the tools needed to rapidly and easily provide their software with a user interface that meets the standards suggested by Commodore-Amiga, Inc. in the AUISG [Com91d].

For more information on the model of applications suggested here, see the rest of this section of the DevCon notes.

For more information about AppBuilder, contact INOVAtronics at:

Inovatronics, Inc.
8499 Greenville Ave, Suite 209B
Dallas, Texas 75231-2499
USA
Phone: 214-340-4991
FAX: 214-340-8514

Bibliography

- [CN91] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, second edition, 1991.
- [Com91a] Commodore-Amiga, Inc. *AMIGA ROM Kernel Reference Manual: Devices*. AMIGA Technical Reference Series. Addison-Wesley, third edition, 1991.
- [Com91b] Commodore-Amiga, Inc. *AMIGA ROM Kernel Reference Manual: Includes and Autodocs*. AMIGA Technical Reference Series. Addison-Wesley, third edition, 1991.
- [Com91c] Commodore-Amiga, Inc. *AMIGA ROM Kernel Reference Manual: Libraries*. AMIGA Technical Reference Series. Addison-Wesley, third edition, 1991.
- [Com91d] Commodore-Amiga, Inc. *AMIGA User Interface Style Guide*. AMIGA Technical Reference Series. Addison-Wesley, 1991.
- [Cor92] GO Corporation. *PenPoint Application Writing Guide*. GO Technical Library. Addison-Wesley, 1992.
- [Tai91] Martin Taillefer. Writing localized applications. In *Amiga Developer Conference Notes*, Denver CO, 1991. Commodore-Amiga, Inc.

1

2

3



Object Oriented Programming

by David Miller

If a builder built buildings the way programmers write programs, the first woodpecker would destroy civilization.

— Anonymous

Introduction

Object-Oriented Programming (OOP). It has easily become one of the most popular buzzwords in the software community in last few years. There are courses on OOP, OO Analysis (OOA), and OO Design (OOD); dozens of books, magazines, and seminars devoted to OO.

What's the fascination?

OOP offers a means of building reusable generic software components.

Imagine what the computer hardware industry would be like if every manufacturer had to make all of their own integrated circuits. In a very real sense, this is the state of much of the software industry today.

How many times have you had to implement a linked list? Or a binary tree? And how many times have you had to stop and draw a diagram to figure out which order to change the references in a doubly linked list to not lose a pointer?

Admittedly, AmigaDOS actually does a better job of managing generic lists than most of its contemporaries. But how do you handle a list of lists? Or a data structure that is really part of more than one list? Perhaps a data structure that describes beef stew and appears on both the animal and vegetable list?

Quick Overview of Object-Oriented Programming

OOP introduces several techniques for dealing with these issues.

One of these is *encapsulation*. Encapsulation is the term used to describe the grouping a set of related data items and providing a well defined interface to the group.

Along with encapsulation, goes *information hiding*. Information hiding occurs when you restrict who or what has access to the data items in the group.

These groups of data items and related functions are referred to as *objects*.

Here is an example of a few objects:

```
GROUP Point
    FUNCTION SetPositionXYZ(NUMBER, NUMBER, NUMBER)
    FUNCTION GetPositionXYZ()

GROUP Dimension
    FUNCTION Set-Size(NUMBER, NUMBER, NUMBER)
    FUNCTION Get-Size()

GROUP Block
    FUNCTION Set-Center(Point)
    FUNCTION Set-Size(Dimension)
    FUNCTION Get-Size()
    FUNCTION Get-Center()
    FUNCTION VisualInfo()
```

Notice that the data elements for these groups have been omitted. This is to illustrate a point about information hiding. Point stores a position given by 3 numbers. The position could be stored as a distance from an origin (cartesian coordinates) or as angles and a distance (polar coordinates), but with information hiding, you don't need to know these details. This implementation describes one pair of interfaces for storing and retrieving the position in cartesian coordinates. Adding direct support for polar coordinates requires just the addition of another pair of interfaces:

```
FUNCTION SetPositionP(NUMBER, NUMBER, NUMBER)
FUNCTION GetPositionP().
```

The Block could have been described by a coordinates for the center and a height and width for the dimension. But by using a Point and a Dimension, Block is not limited to a 3-dimensional cartesian coordinate system. For example, Point can be changed to use polar notation without affecting Block. Likewise, Dimension can be changed to include mass without requiring any changes to Block.

A function that is passed a Block, may obtain the (x,y,z) coordinates by asking Block for the point at its center with Get-Center(). Then it can obtain the coordinates by asking Point for its value in cartesian coordinates.

This is what is called *data abstraction*. Data abstraction shifts the focus from:

“What am I doing with this data?”
to
“What is the data that I am manipulating?”

Another useful tool of OOP is *inheritance*. Inheritance allows the programmer to create a variation of an existing object without writing it from scratch. To create a Block that has a color, this is all that is needed:

```
GROUP ColorBlock
    INHERIT    Block
    FUNCTION   SetColor(Color)          FUNCTION   GetColor()
    FUNCTION   VisualInfo()
```

ColorBlock will use the same dimension and positioning routines as Block. From ColorBlock, the programmer might create still other types of Blocks:

```
GROUP MetalBlock
    INHERIT    ColorBlock
    FUNCTION   SetMetal(Metal)          FUNCTION   GetMetal()
    FUNCTION   VisualInfo()

GROUP WoodBlock
    INHERIT    ColorBlock
    FUNCTION   SetGrain(Grain)          FUNCTION   GetGrain()
    FUNCTION   VisualInfo()
```

If all of these groups were to describe objects for a rendering system, it could become very cumbersome for the rendering program to need to know about every form of a Block. For this, OOP uses what is called *polymorphism*.

As far as the rendering system is concerned, there is only a Block. And a Block has a function called VisualInfo() which returns some data to the rendering system about the appearance of the Block. The children of Block redefine VisualInfo() to return appropriate descriptions.

Polymorphism is the ability to reference an object whose type may only be determined at runtime. The rendering system can treat all of the various children of block as being type block and trust that the correct VisualInfo() will be called.

Benefits of Object Orientation or, "What's in it for me?"

To establish how OOP can help the software engineer, it's best to begin by establishing some software engineering goals or concerns and seeing how OO can help realize these goals.

☐ **Correct/verifiable**

Software needs to be correct and it is desirable to have the ability to prove that it is correct. Attempting to prove the correctness of a system is impossible if you cannot prove the correctness of each of its parts. OO will not automatically provide you with a provably correct system. It does however provide a means of addressing correctness at the component level through the use of assertions. Assertions generally come in three flavors: preconditions, postconditions, and invariants. A precondition is an initial requirement on entry to a routine (e.g., "intuition.library" must be open before calling OpenScreen()). Postconditions provide a check that the routine has achieved its goal (e.g., did OpenScreen() return a non-NULL pointer?). Finally, invariants provide a set of boundaries that the object agrees not to violate. For example, a routine that manipulates an 16-color screen may have an invariant that requires the number of a bitplane to fall in the range of 0 to 3.

☐ **Compatible (with similar products)**

The more popular of today's OOPLs generally use C as an intermediate language. While in a pure OOPL, some might consider it tantamount to heresy to link in procedural elements, in practice C and assembler are still often used for performance reasons. Over time, as the compilation systems continue to develop and execution environments become faster and more complex, this will become less necessary and may eventually cease altogether. (In particular, C++ and Objective-C, being closely related to C may eliminate the need for "pure" C code. And, it is conceivable that CPUs will one day reach a level of sophistication in instruction and data pipelining that it will be beyond the capability of a human being to write assembly code that rivals the compiler generated code.)

☐ **Efficient**

The ability to link with impure languages such as C and assembler provides a short term means of writing highly efficient code. In time, as compilers and machines advance, this will become a non-issue.

☐ **Portable**

By using data abstraction, it is possible to isolate the machine, hardware, or operating system dependencies to a few objects. These objects may serve as ancestors to new objects that will implement the interface on a new platform.

☐ **Be easy to use**

There is a great deal of ongoing research into ways of simplifying the development process. Some languages include sophisticated class browsing tools that allow the developer to observe the interactions between classes and examine the interfaces defined.

☐ **Maintenance is a large percentage of software engineering life cycle**

Some estimates place the percentage as high as 70%. As explained above in the discussion of reusability and extensibility, OO allows testing to be more focused and in theory will reduce testing time and development time. Additionally, polymorphism easily permits the extension of an application in ways that were not conceived of during the original development.

☐ **Robust**

(with respect to abnormal cases)

☐ **Extendible**

(with respect to changes/modifications, important for large-scale programs)

☐ **Reusable**

☐ **Have integrity**

OO has the ability to address the last four issues through encapsulation, information hiding, inheritance and polymorphism. Consider the difference in adding support for a new account type to first, procedural code, then to OO code.

Procedural

```
if accounttype == Savings
    SavingsProcessing();
...
```

Add this every place it switches based on the account type:

```
else if accounttype == MoneyMarket
    MoneyMarketProcessing();
```

OOPL

```
OBJECT Account
    FUNCTION deposit();
```

```
OBJECT Savings
    INHERIT Account
```

Simply derive a new object that redefines the deposit routine. No other changes are required...

```
OBJECT MoneyMarket
    INHERIT Account
    REDEFINE deposit
        ... require initial deposit to be > $1000
```

because the code handling deposit transactions calls the accounts deposit function, which in turn resolves to the redefined deposit function when the account is a MoneyMarket account.

```
account.deposit();
```

The interface has been extended to support a new object without altering any of the code in the existing interface. By limiting the scope of code changes in this manner, the risk of adversely affecting the system as a whole is minimized. Since the new functionality does not alter the existing interfaces, if the new account is proven to comply with its assertions, the resulting system may be assumed to be proven. In terms of actual development and testing, OOP has removed the need to exhaustively test the existing functionality.

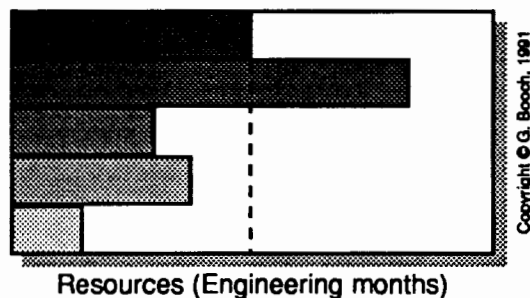
Beware, there are certain assumptions in operation! The account object and the object that has contracted for its services must be in agreement on the interface, including the handling of error conditions (which may occur if an attempt is made to make an initial deposit of less than \$1000 to a money market account).

OO Development

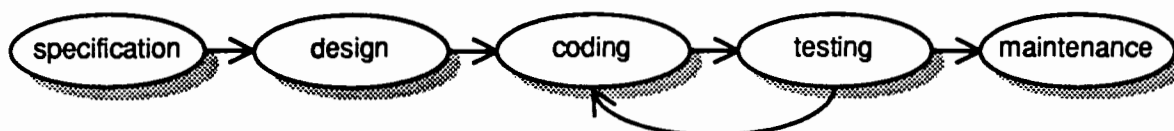
A word to the newcomers... If you are a C programmer who is interested in learning C++ (and to a lesser degree this is true for Objective-C) it is suggested you begin by learning Smalltalk or Eiffel first. By using a language that is not closely related to C you prevent yourself from slipping back to non-OO programming practices. If this is not an option, taking a good course in OOP with an emphasis in the language you have selected is strongly recommended.

Changing Expectations:

Analysis
60-70% more design effort
40% less coding effort
25% less testing
70% less effort in system

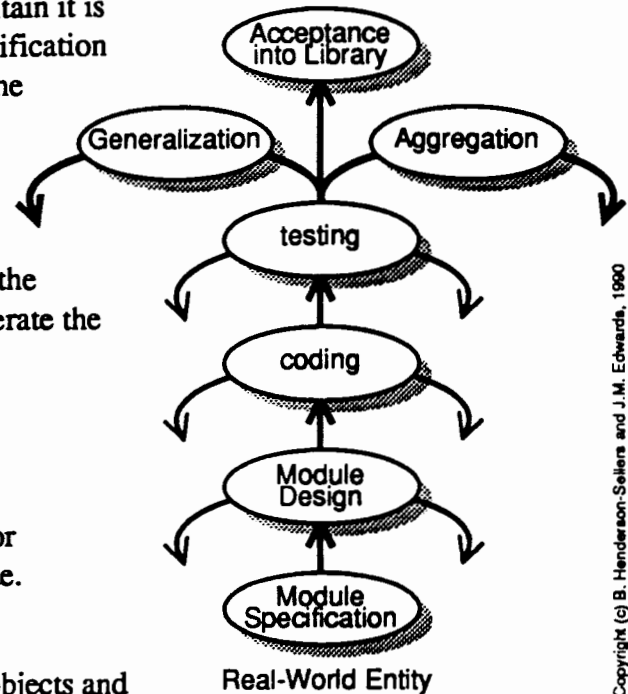


Traditional view of Development Life Cycle



OO Development Life Cycle "Fountain"

Take special note that at any point in the fountain it is possible to return all the way back to the specification phase! There is no implicit assumption that the specification is correct as there often is in the traditional, procedural design process. This is not to say that the traditional approach is wrong for procedural programming. Rather, it is typically much easier to enumerate what the desired actions of a system than it is to enumerate the objects involved and their interrelationships.



Copyright (c) B. Henderson-Sellers and J.M. Edwards, 1990

Phases of OOA

- ☐ **Specify systems requirements**
Search the customer's requirements for "things" and the services they provide.
- ☐ **Identify the objects/classes**
Which identified "things" constitute objects and which constitute attributes of objects?
- ☐ **Identify the relationships**
Generalization: this is a _____.
Association: this uses _____.
Aggregation: this has a _____.
- ☐ **Design the relationships**
Define what services each object offers and what services it requires.
- ☐ **Look in existing library for appropriate objects**
Does a class exist that represents this object or could serve as a parent for this class?
- ☐ **Look for inheritance relationships**
What elements are in common between the objects? Do these elements form the basis of a common parent object?
- ☐ **Generalize**
Can the new objects be generalized? What concepts offer potential for reusability?

Repeat until the class(es) are acceptable for use as a library

Step 1

Systems requirements specification (SRS)

- in language of users
- source documents to find objects

Step 2

Find candidate objects
(real-world objects)

- First pass -- nouns in SRS
 - + verb methods
 - + adjective attributes
- Concrete *and* abstract
- (Coad & Yourdon, 1990; Nerson, 1990)
 - e.g., Structure and classification
 - Events
 - Roles played
 - Location
 - Organization

Copyright (c) B. Henderson-Sellers, 1991

Step 3:

Establish Interactions

Analysis relationships: Classification
Association, Aggregation

Step 4:

Analysis merges to Design

Design relationships: Client-server
(with current OOPLs)

Step 5

Explore Existing Library Classes

Refine Design -- now highly detailed

Step 6

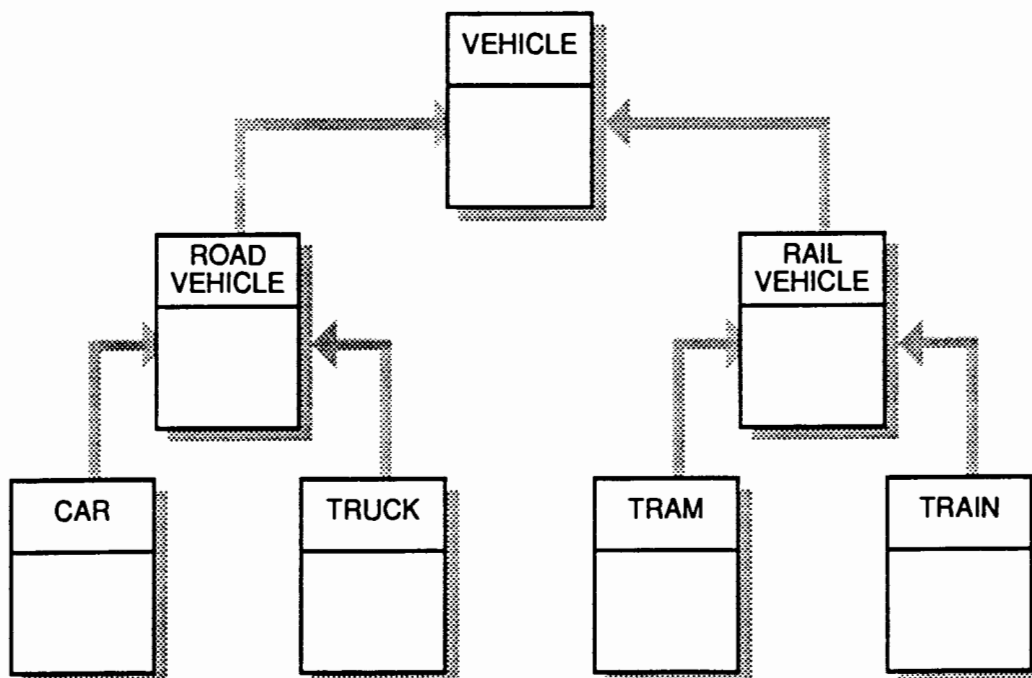
Examine Class Network for more
Inheritance Structures

This may introduce new classes and
new interactions

Class Coded and Tested

Copyright (c) B. Henderson-Sellers, 1991

OFF-LINE INHERITANCE HIERARCHY



Copyright (c) B. Henderson-Sellers, 1991

Step 7

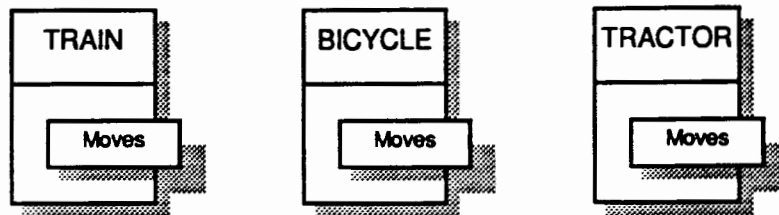
Further Refinement beyond demands of current project in order to facilitate later reuse

Cluster Identification

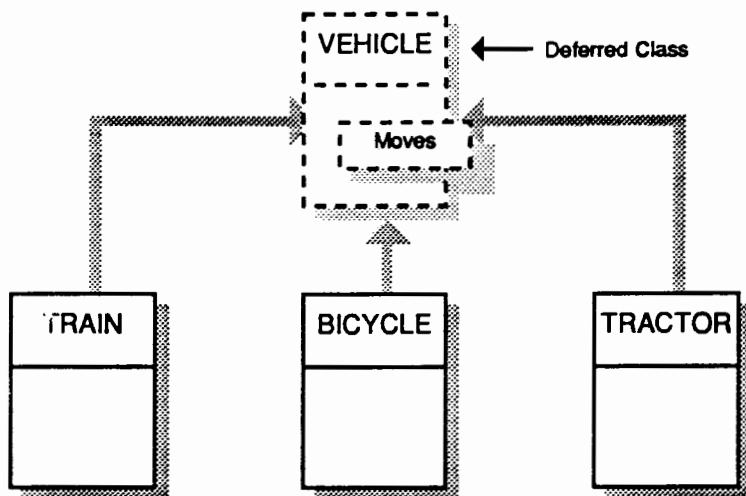
Documentation

Copyright (c) B. Henderson-Sellers, 1991

WE FIND WE HAVE DESIGNED THE CLASSES:



GENERALIZE TO



Copyright (c) B. Henderson-Sellers, 1991

Methodology

Step 1. Requirements Specification

e.g., customers deposit or withdraw cash or checks into one or more accounts. Available accounts are:

- a) Checking Account
- b) Passbook Account
- c) Savings Investment Account
- d) Security Plus Investment Account
- e) Term Deposit Account
- f) Short-Term Call Account

Copyright (c) B. Henderson-Sellers, 1991

Step 2. Identify Entities

Fairly easy:

- 1) Customer
- 2) Checking Account (A)
- 3) Passbook Account (B)
- 4) Savings Investment Account (C)
- 5) Security Plus Investment Account (D)
- 6) Term Deposit Account (E)
- 7) Short-Term Call Account (F)

But what about:

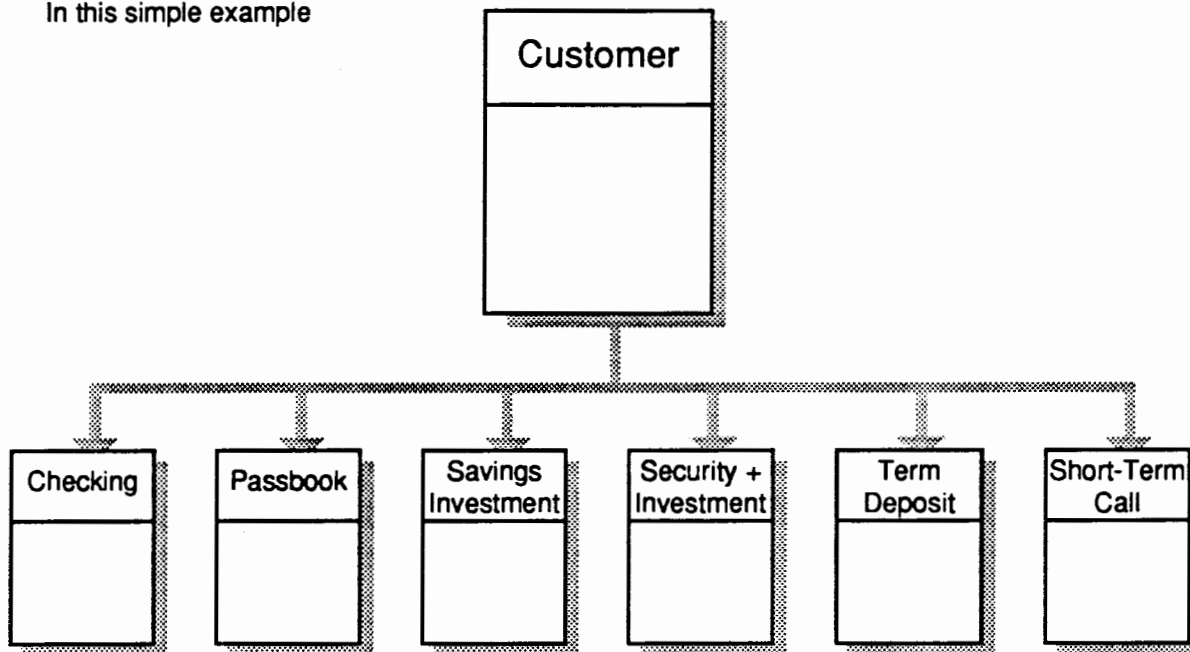
- 8) Cash
- 9) Check

These could be attributes or objects in their own right

Copyright (c) B. Henderson-Sellers, 1991

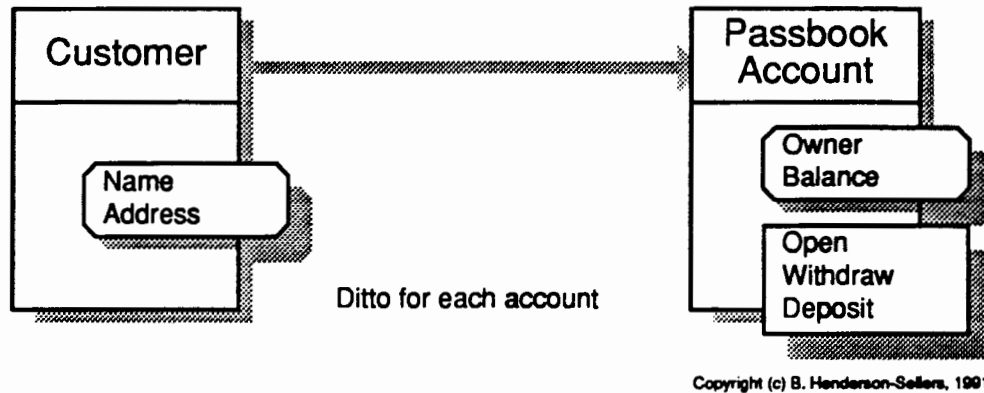
Step 3. Establish Interactions

In this simple example

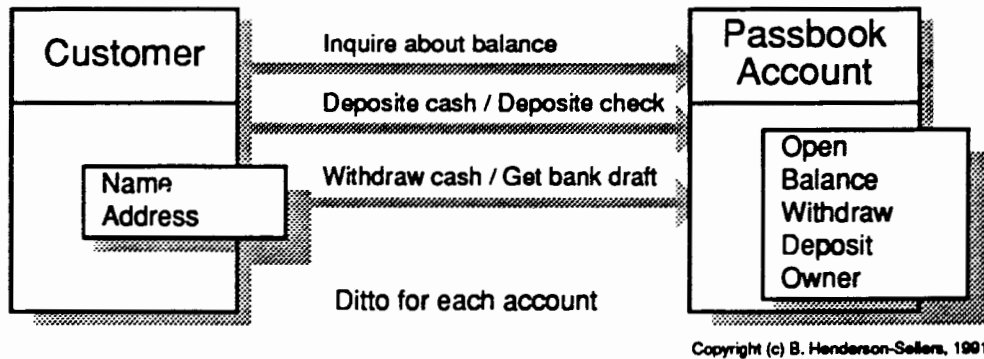


Copyright (c) B. Henderson-Sellers, 1991

Add Attributes and Operations



Step 4. More Detailed Design



Step 5. (Not really here, since starting from scratch)

We might look for library classes to represent

- a) case as a REAL
- b) account_name as a STRING
- c) possibly address as several fields (an "address" class)

Copyright (c) B. Henderson-Sellers, 1991

Step 6. Look for Inheritance Hierarchies

Required knowledge

- No interest on Checking Accounts. Must maintain minimum \$250 balance.
- Passbook. No minimum balance. Interest rate dependent on balance.
- Savings Investment. Minimum balance \$500. Withdrawals must be \$100 or more. Interest rate dependent on balance.
- Security Plus Investment. Minimum balance \$5,000. Minimum transaction \$500. Interest rate dependent on balance.
- Term Deposits. Fixed term. Minimum deposit \$500. Interest rate depends on both term and deposit.
- Short-Term Call Account. Minimum deposit \$10,000. Minimum period of deposit 7 days. All transactions minimum \$1,000. Interest calculated daily (single rate).

Copyright (c) B. Henderson-Sellers, 1991

Table of Interest Rates

Passbook

\$1 - \$1,999	10%
\$10,000 - \$19,999	13%
\$2,000 - \$9,999	12%
>= \$20,000	15%

Savings Investment

\$500 - \$1,999	9.5%
\$2,000 - \$9,999	12%
\$10,000 - \$19,999	13%
>= \$20,000	14%

Security Plus Investment

\$5,000 - \$9,999	12%
\$10,000 - \$19,999	13%
>= \$20,000	15%

Term Deposits

Term	\$500-\$49,999	>= \$50,000
1 mth to <3 mths	15%	15.5%
3 mths to <6 mths	16%	16%
6 mths to <12 mths	16.5%	16.5%
12 mths to <13 mths	17%	17%
13 mths to <33 mths	18%	18%
33 mths to <60 mths	18%	18%

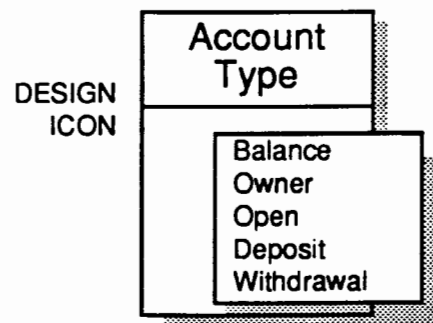
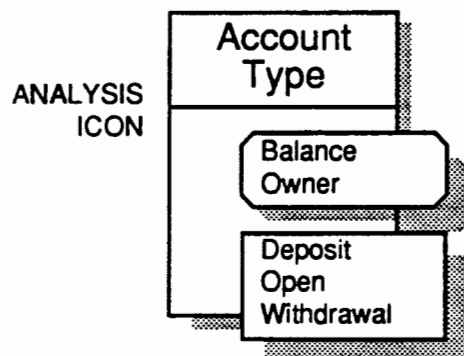
Short-Term Call Account

11%

Copyright (c) B. Henderson-Sellers, 1991

One Suggested Route

Construct an object class for each type of account



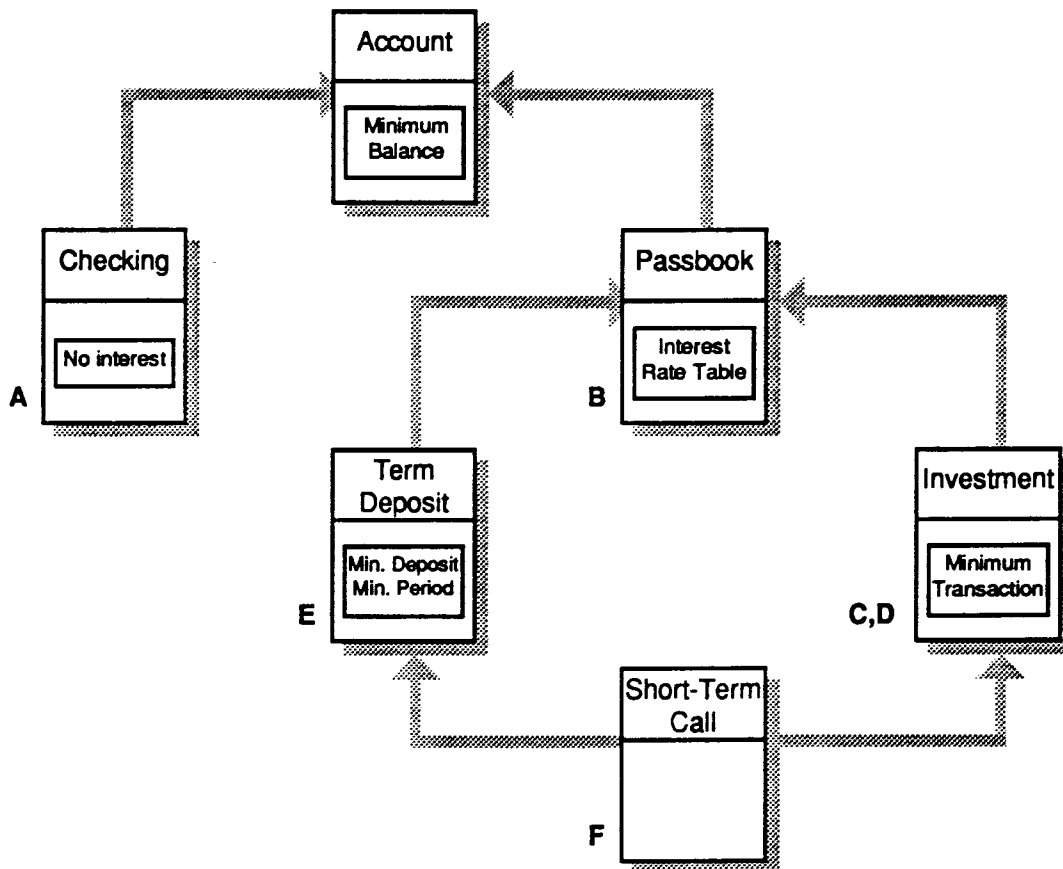
Copyright (c) B. Henderson-Sellers, 1991

Recognize commonality

		Interest	Flat Rate Interest	Interest Table	Value of Min. Transaction	Value of Min. Transaction	Min. Deposit	Min. Period of Deposit
A	Checking				250			
B	Passbook	✓		✓				
C	Savings Investment	✓		✓	500	100		
D	Security Plus Investment	✓		✓	5,000	500		
E	Term Deposits	✓		✓	(500)		500	✓
F	Short-Term Call Account	✓	✓		(10,000)	1,000	10,000	✓

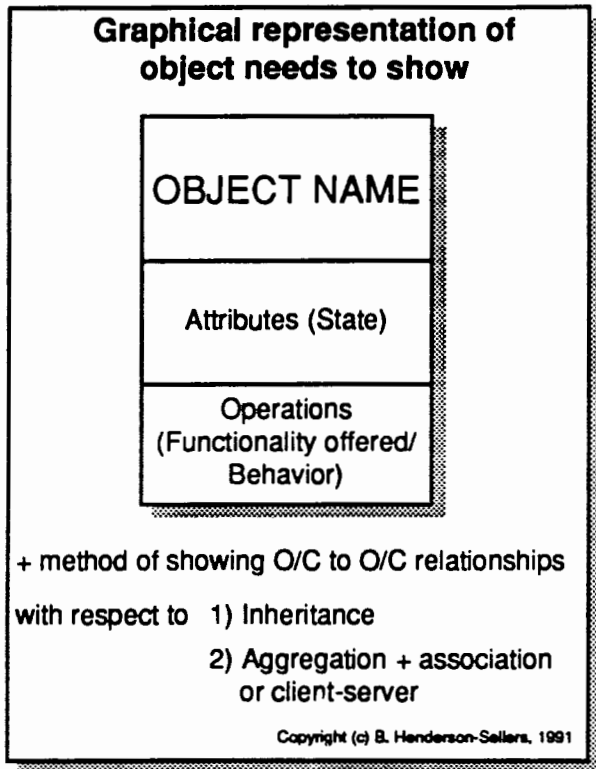
Copyright (c) B. Henderson-Sellers, 1991

Generalization

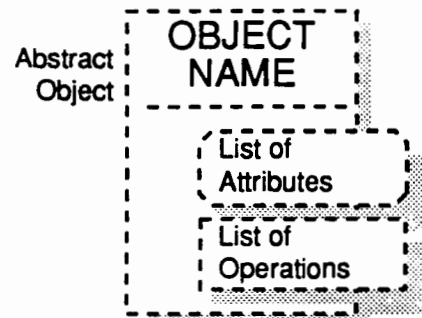
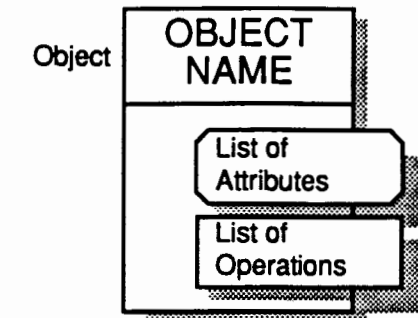


Copyright (c) B. Henderson-Sellers, 1991

A Useful Notation for OO Analysis and Design

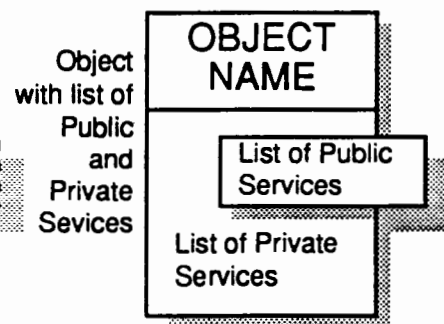
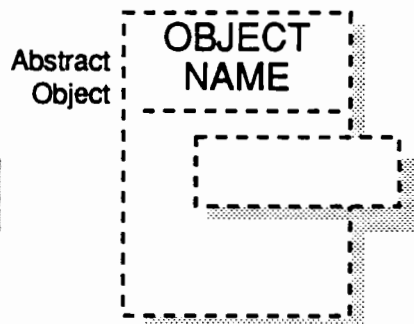
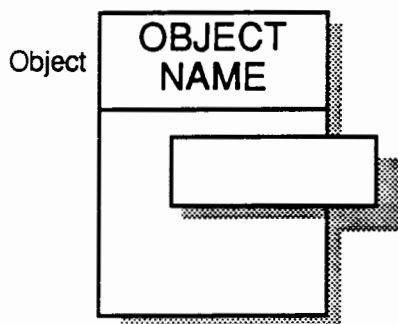


ICONS FOR ANALYSIS



Copyright (c) J.M. Edwards and B. Henderson-Sellers, 1991

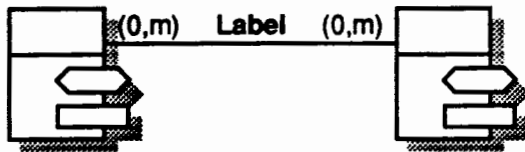
ICONS FOR DESIGN



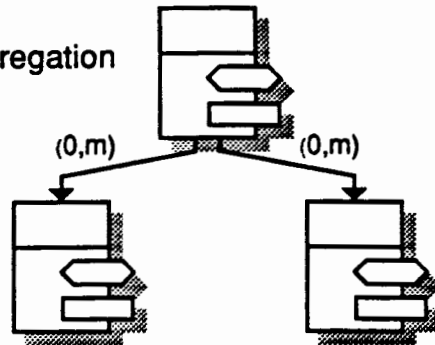
Copyright (c) J.M. Edwards and B. Henderson-Sellers, 1991

Analysis

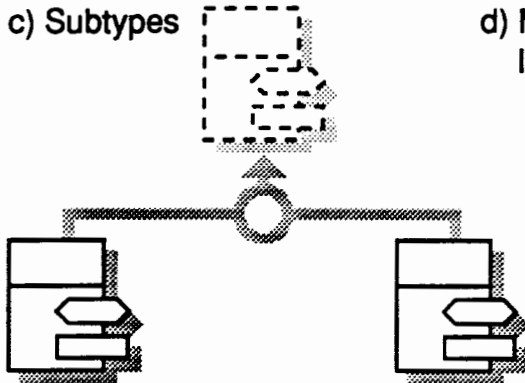
a) Association



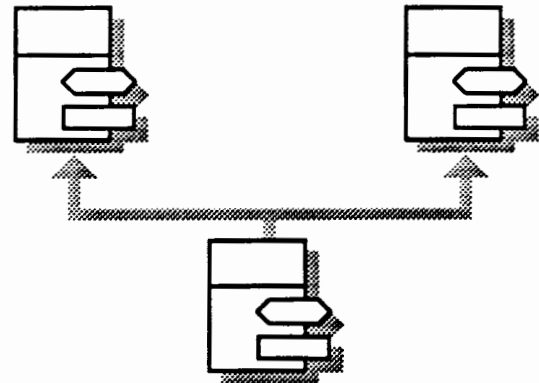
b) Aggregation



c) Subtypes

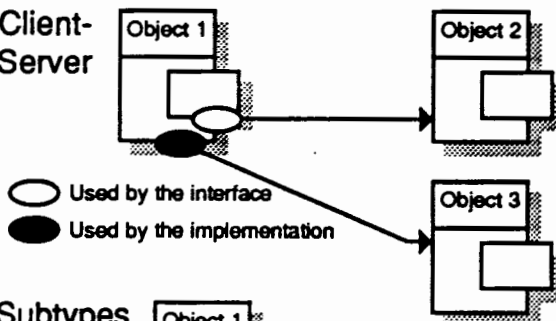


d) Multiple Inheritance

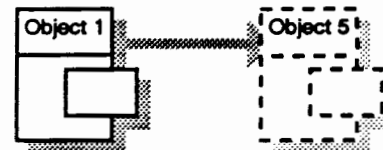


Design

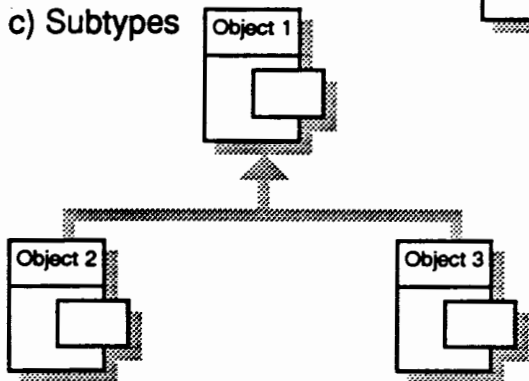
a) Client-Server



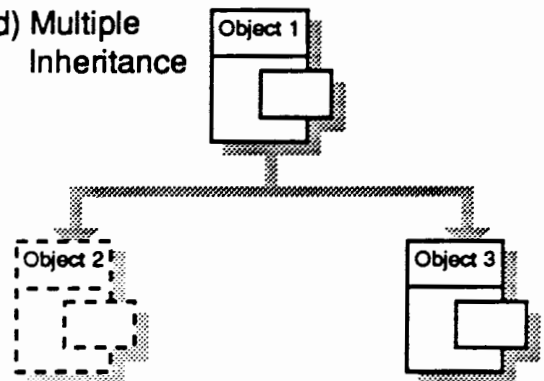
b) Single Inheritance



c) Subtypes



d) Multiple Inheritance



Alternative approaches to Analysis and Design

While a purely OO approach is desirable, it is not always possible. It is however possible to develop using OO techniques from a non-OO specification. Likewise, it is possible to carry out the full OOA/OOD process but then implement the software using a functional language such as C.

O-O-F	O-O-O	F-O-O
1. Object-oriented systems requirements specification ⋮		Functional systems requirements specification
2. Identify objects		Draw DFDs
3. Establish interactions		Semantic Data modeling
4. Lower-level detailed design		Transform to lower-level detailed design
5. Use of libraries and simulation of inheritance	Bottom-up use of library classes	
6. Revision to transform into a design compatible with procedural code	Add inheritance hierarchies	
7. Code using procedural modules	Aggregation/generalization	

O-O-F refers to OO analysis and design followed by a functional programming implementation.

O-O-O is the pure Object-Oriented approach to software development.

F-O-O describes OO design and implementation from a functional specification.

Another useful technique is the use of CRC cards.

CRC = Class, Responsibility, Collaboration

A technique for the earliest stage of the design process, CRC cards provide a very simple mechanism for exploring relationships between objects and identifying key objects, collaborations, and systems.

This is primarily intended as a group exercise in the style of brainstorming. CRC cards are a useful exploratory tool, but are not really suitable for the later design work.

Summary

Object-oriented programming is not the answer to all the woes of the software world. And simply switching to an object oriented programming language will not magically make your code more maintainable. Object-orientation is not a programming language or a different way of analyzing a problem. OO is a philosophy. It requires a new mindset, new approaches, new timelines. Design and analysis become the focus instead of the implementation. So take the time to do it right, learn the new ways of thinking. The potential long-term savings are significant.





Product Testing

by John Kominetz

Testing is essential for creating a successful product in today's computer-savvy market. Users do not respond kindly to visits by the Guru anymore, and professionals won't tolerate losing a day's work to a software error. Testing builds quality software ; quality inspires consumer confidence and trust; consumer trust leads to sales and financial gain for the developer.

In a tight economy, harsh deadlines and staff reductions often weaken the testing process, which is rarely seen as integral to the development process. Hopefully the following will help improve the internal testing process and provide an inexpensive alternative to complement it.

Beta Testing

Harrison's *Improving the Software Testing Process* defines testing as "a systematic exercise of system components to find and classify errors with a minimum of time and effort." Testing should be an orderly, well-planned process much like a scientific experiment. Its only goal is to find and classify errors. Testing should reduce development time and maintenance releases by getting the problems out early.

Most traditional texts on testing say the same thing; I find this definition is too narrow, however. Beta testing--usually "seat-of-the-pants" testing--contributes many error reports without rigorous procedures; varied expertise, systems environments, and tester enthusiasm compensate for the formless nature of it. Beta testing is not a substitute for rigorous system testing. Rather, it complements and enhances it.

Testing is more than finding and classifying errors. It is often the conscience of the development process, reminding those under pressure to produce that quality cannot be overlooked for the sake of a deadline. Quality and reliability are the strong foundation on which to build a good product concept. Without this foundation, the greatest product concept will not sell if it crashes or loses data with alarming frequency.

Beta testing is any testing outside the development test process. It often involves personnel not employed by the developer. Beta testers can employ whatever testing methods they wish--from regimented test-case planning to just playing around; they rarely submit test plans or test logs, and their only real requirement is to submit error reports.

Advantages of Beta Testing

Relatively speaking, beta testers are free. The biggest expenses are finding the beta testers, distributing updates to them, collecting reports from them, and compensating them for their efforts. Beta testing is more a hobby than a job; do not expect to draw a salary as a beta tester, nor spend 40 hours a week beta testing.

Beta Testers can range from inexperienced users to professionals using the product in their businesses. Product assurance or testing departments tend to employ technical people, and simple errors can slip by such departments because of assumptions new users don't have, like trying to load a GIF file into a spreadsheet program. Aside from experience, beta testing allows access to many hardware and software system variants that the developer may not have. Users have old motherboards, strange CPU add-ons, outdated operating systems, emulators, odd things in their startup-sequences, etc. Amigas in particular are blessed with multi-tasking without memory protection, making it easy for interaction between applications. Further, consider how many things can be stuck into an A500 expansion slot or even the memory expansion slot!

Some would argue that many of these products--odd accelerators or unusual programs--cause the many errors reported by beta testers, not the developer's own project. When in doubt, the user blames whatever application has the error requester on it. A few lines in the manual about known incompatibilities may save users from conflicts ahead of time, or at least let them know who is really at fault.

Disadvantages of Beta Testing

There are two major disadvantages to beta testing: First, as more people not involved with the development process obtain the product, the likelihood of piracy increases. Sometimes companies will issue unique serial numbers as part of their product/beta test package so they can track leaks, but this is time consuming. Second, without direct control over the tester's procedures, it is impossible to know the quality of testing being done. Ultimately, only the number of legitimate error reports shows how effective a beta tester is. Be prepared for some testers that submit no error reports because they lost interest. This second reason is why regular testing is critical to the development process; without it there is no way to get a feel for how well a product has been tested.

Choosing Beta Testers

The defining characteristic of a good beta tester is interest in the product so he will spend valuable free time testing it. Experience and exotic equipment can be useful, but the tester

must spend time testing the product to find errors. There are four broad classes of beta testers, each with advantages and disadvantages, but they all require the basic enthusiasm or need to test the product to be good testers. The four types are as follows:

Company Employees

If the company is large enough, it is possible to get people not associated with the development process to use the pre-release product. Most companies already have non-disclosure agreements with their employees, eliminating some of the legal mumbo jumbo required to use non-employees. Upgrading a version of the beta software can be as easy as leaving a disk on someone's chair. Getting or clarifying error reports is similarly easy when the person is accessible for 40 hours per week.

Users

The bulk of beta testers come from common every-day users, preferably the kinds of users that would buy the product. Even with non-disclosure agreements, this is how many beta versions of a product find their way to pirate bulletin boards. Upgrading and collecting reports can be difficult with this class of tester. Worse, the user probably has a job, maybe even a spouse or family, reducing the amount of time available for testing the product. Still, through friends and user groups, these are the most accessible candidates for beta testers.

Other Developers

Other developers in a similar but non-competitive market can provide more sophisticated testing than the average user. If the beta product is designed to cooperate with another developer's product, behind-the-scenes information can uncover errors quickly. The only problem with using other developers is that they are usually as manpower-short or deadline-weary as you are! Approaching individuals in other companies can give you a user-style beta tester with exceptional technical knowledge. Be careful that beta testing your product does not constitute a conflict of interest for the tester!

Professional Users

This is the most productive, and potentially the most annoying, candidate for beta testing. Seek out companies that could use your product as part of their business. Nobody will stress a product as much as somebody using it 40 hours a week. Here, the tester's interest is replaced by something far more demanding--the user's needs. Unlike normal beta testers, this class of tester will be eager to report errors, and even more eager to receive newer versions with bug fixes. Be prepared for many phone calls. Also be prepared to provide rapid technical support, workarounds, and quick-fix versions. While this can be the most effective kind of beta tester, expect him to demand more support and interaction from you.

Getting Started

To weed out beta test candidates, require them to fax or write an official request to become a beta tester. Make sure the letter states that upon receiving the initial beta test packet, he agrees to be bound by a non-disclosure agreement. Marginally interested candidates will not be bothered with such "formalities." These are often the people who cannot be bothered with submitting reproducible error reports or other such "formalities."

Once the letter is received, send the tester a complete beta test package including the product, current documentation, release notes, a non-disclosure agreement--to be signed and returned immediately, and information on error reporting. The information on error reporting should include a standardized error report form, guidelines for writing a reproducible error report, and information on who to contact with error reports and questions. Preparing a brief guide to testing and reporting errors can save much time and effort, especially if it eliminates the need to clarify ambiguous reports.

Depending on how often new versions are available, a beta tester should get a complete test package, sans non-disclosure agreement, with each new beta release. This does not mean that you should send testers every new version of the product; designate special landmark builds as new beta test material--maybe three or four during the entire development process.

Compensating Beta Testers

It is not unusual to offer extra incentives to beta testers. The most common form is to give the tester a copy of the finished product. Having a prize for the individual with the most reproducible errors can build a healthy feeling of competition among the testers, especially if the top three scores are distributed with the latest test package. Often, having early access and direct technical support for a new product is enough to satisfy most beta testers. Use whatever feels most comfortable.

More important, keep the names of effective beta testers for future product testing, and share this information with other developers. Develop a good database of testers, and you will have a valuable resource without investing all the start-up costs each time.

Conclusion

Beta testing can be an inexpensive and effective form of alternative testing. Careful preparation and research on your part are required, but the result is a higher quality product. Beta testers can become much more than just testers, even as a source for new employees or business alliances. Remember that beta testing itself is not enough, and that rigorous product testing as part of the development process is still required.





Using Amiga Debugging Tools

by Carolyn Scheppner & Adam Levin-Delson

Have you ever experienced any of the following problems with software ?

The software runs well on your system but some others report it has problems on their systems.

The software runs well by itself but has problems running with or after other software.

The software runs well most of the time but occasionally crashes or fails for no apparent reason.

You can find and eliminate many of these types of hidden software problems by running Amiga debugging and stressing tools while developing and testing your product.

Hidden and obvious software problems are often caused by use of null pointers, uninitialized pointers, improperly initialized structures, improper use of I/O requests, improper abort code, improper memory usage, or overwriting of memory allocations. By using Enforcer, Mungwall, IO_Torture, Scratch, Memoration, and other Amiga debugging and testing tools, you can catch most of these problems before you ship a product.

In fact, many companies now require that all of their in-house software pass at least Enforcer and Mungwall testing, and have also added this requirement to their contracts for outside development. Many other CATS tools such as Devmon, Owner, and LVO can help to pinpoint the causes of problems. You have these tools. Here are hints on how to use them, when to use them, and how to get the most from their output.

Enforcer and Mungwall

Enforcer and Mungwall are the top two bug finding and bug prevention tools. Enforcer is an MMU-based debugging tool by Michael Sinz, based on an earlier tool by Bryce Nesbitt.

An MMU is a memory management unit which can be configured to trap accesses to specified ranges of memory. The 68030 and 68040 processors have a built-in MMU, and most 68020 boards contain a separate MMU. Because it is MMU-based, Enforcer can trap reads and writes of low memory and non-existent memory the instant these accesses (also known as "Enforcer hits") occur. This allows you to catch usage of null pointers and some

uninitialized pointers in your program, and even accesses which would have trashed low memory or otherwise crashed the system.

Some of these accesses may seem harmless on your system (such as reads of address 0) but could cause your program to fail in the field. If you are developing commercial software (or any software that you plan to distribute) it is extremely important that you invest in a MMU, Enforcer, and Mungwall. As more of the development community begins running these tools, software that is unusable in their presence will simply not be used. The main differences between the new Enforcer and the previous Enforcers is that the new Enforcer also works with 68040 processors and has a wide variety of output options built in. In addition, due to the variety of output options, the new Enforcer must be *run*. The new Enforcer requires at least V37 OS, so if you need to run Enforcer on a 1.3 machine, you must use the older Enforcer 2.8b which we will probably be renaming *Enforcer1.3*. Full docs are provided with Enforcer.

Enforcer is even more powerful when used in combination with Mungwall. Mungwall is a combination memory munging tool by Ewout Walraven which is based on Memmung by Bryce Nesbitt and Memwall by Randell Jesup. The “mung” part of Mungwall fills all of free memory (and all subsequently freed memory) with a nasty odd 32-bit values like \$DEADF00D. Such a values are almost guaranteed to cause serious problems for any program that uses uninitialized pointers or structures, or uses memory or allocations after they are freed. Such usages can occur, for example, when allocations are not freed in the correct order.

Mungwall uses a few different nasty 32-bit values in its memory munging to help you diagnose any problems.

- ☐ Except when Enforcer is running, location 0 is set to \$CODEDBAD so that programs referencing location zero will not find a value. Programs referencing location 0 as a string will get a string of high ASCII characters rather than a null string, and programs using null structure pointers should be irritated into crashing. When Enforcer is running, this is not necessary because with location 0 containing 0, Enforcer can trap these low memory accesses by itself.
- ☐ On startup all free memory is munged with \$ABADCAFE. If this number shows up, someone is referencing memory in the free list.
- ☐ Except when MEMF_CLEAR is set, memory is pre-munged on allocation with \$DEADFOOD. When this is seen in an Enforcer report, the caller is allocating memory and doesn't initialize it before using it.

- ☐ **Memory is filled with \$DEADBEEF when it is deallocated, encouraging programs reusing freed memory to crash or get Enforcer hits.**

The “wall” part of Mungwall allocates extra memory before and after every memory allocation and fills this “wall” with a fill pattern and other information. On each de-allocation, Mungwall checks to make sure that the deallocation size matches the size of the allocation, and checks to make sure that the walls have not been overwritten. Mungwall also watches for 0-size allocations, 0-size deallocations, and 0-address deallocations. Mungwall checks for incorrect allocations (such as 0 size) during AllocMem, and checks for incorrect deallocations or trashing around allocations during FreeMem. If trashing or incorrect deallocation is detected, Mungwall will both report it and refuse to free the memory. These reports are all known as “Mungwall hits”.

The latest Mungwall can also optionally report on failed allocations with the SHOWFAIL option. In addition, Mungwall has an option to “snoop” and report on all memory allocations and deallocations for all tasks or specified tasks. This can be useful when tracking down memory losses. The sometimes voluminous snoop output can be run through the snoopstrip program which will throw away all matching allocation/deallocation pairs.

Another useful recently added Mungwall feature is the NAMETAG option. When this option is active, Mungwall will tag all memory allocations with the first 16 characters of the name of the task, process, or command that allocated the memory. A program called Munglist can then do its best to show the names of the allocators of currently allocated pieces of memory. This can be very useful in tracking down memory losses. However, keep in mind that some of an application’s allocations may be done by a different system task (for example, if an application opens a file, a file handle or buffers may be allocated by the process for the disk volume).

Mungwall may be used without Enforcer and on non-MMU machines. If you don’t have an MMU, at least test with Mungwall alone. If you are using uninitialized memory or memory after it is freed, Mungwall should help to crash you immediately (as you might crash on a user’s machine when he runs other programs at the same time as yours). Mungwall is more pleasant when used with Enforcer however, since it will generally incite an Enforcer hit when memory is misused, rather than a crash.

It is generally not safe or recommended to turn Mungwall off and back on without rebooting. This is because Mungwall recognizes its walled allocations by looking for a special value it has placed near the beginning of the allocation. If a program allocates memory while Mungwall is running, then frees it while Mungwall is turned off, and another program reallocates the some of the same memory while Mungwall is turned off, and then tries to deallocate it with Mungwall turned back on, Mungwall may generate Mungwall hits due to

incorrect data. Note that this can also occasionally happen on a soft-reboot if a task that starts up and allocates memory before Mungwall is run happens to receive a Mungwall-tagged piece of memory that was in use before reboot, and then frees the memory after Mungwall is running. To identify such problems, turn your machine off for 15 seconds or so to clear memory, turn it back on, and retest.

A useful alternative to turning Mungwall off and on is the Mungwall UPDATE feature. UPDATE allows you to turn on and off many of the options of another running copy of Mungwall. For example, you could turn off the MEGASTACK option via Mungwall UPDATE NOMEGASTACK or turn on the SHOWFAIL option with Mungwall UPDATE SHOWFAIL.

Debugging Terminals

Enforcer and Mungwall both output their debugging information to the serial port at the baud rate your machine's serial hardware is set to. At powerup, your serial hardware is set to 9600 baud, but you can modify this by bringing up a terminal package and setting a baud rate. The optimal debugging setup is to connect your Amiga via a null modem serial cable to another Amiga or computer running a terminal package with ACSII capture capability. Both Enforcer and Mungwall include Ctrl-Gs in their output to generate a beep with most terminal packages, and the ASCII capture capability will allow you to capture all serial debugging output to a file for examination. This is especially useful when combined with serial `kprintf()` (debug.lib) debugging statements in your code such as `kprintf("About to close window $%lx",win)`.

But some developers don't have a second machine to use as a debugging terminal, or may need to debug a serial application which would interfere with serial debugging output. So two other options exist. One is to use the parallel versions of the debugging tools, and the parallel debugging `dprintf` command (from `ddebug.lib`) and connect a parallel printer for the output. The benefit of serial and parallel debugging is that you can get some information out and capture it even if your machine is crashing, and even if your application has taken over the system. The other option is *Sushi*.

Sushi

For most applications, an option now exists for debugging on a single machine. It's called *Sushi* (a late-night name loosely based on the fact that it captures "raw" serial output). Sushi captures all raw serial debugging output (such as `kprintf()`, new Enforcer with RAWIO option, Mungwall, IO_Torture, etc.) and has its AmigaDOS process send the debug information to stdout. This means you can redirect Sushi's output to a an AUTO/CON

window (and the window will pop open if you have any hits), or to a multiseriial port, etc. Sushi can also be told to save its circular buffer (via Ctrl-F signal or via a separate SUSHI SAVEAS FILENAME command), or to empty its buffer (Ctrl-E signal or a separate SUSHI EMPTY command).

Sushi also can operate in quiet mode, just capturing debugging output, and provides a program task signalling interface which allows an external application to be written to display the captured debugging output (for those of you who want a fancier debugging display program). A benefit of Sushi is that fairly voluminous debugging output can be used even from within interrupt code or intense task code because Sushi only has to place the text bytes in a ram buffer, and the actual output is done by a separate process. See Sushi.doc for more information and a source for a sample external display program.

Sample user-startup:

```
run >NIL: Mungwall
run >NIL: Enforcer RAWIO
run >NIL: sushi <>"CON:0/20/640/100/Sushi CTRL-File CTRL-Empty/AUTO/CLOSE/WAIT" NOPROMPT
```

Debugging printf()s: kprintf() and dprintf()

When using serial (or parallel) debugging and stress-testing tools, it is very useful to have your own program's debugging statements (such as "About to do xxx") mixed in with any potential hits from the debugging tools.

In addition, it is very useful to have a printf()-like command that can be used from within even the low level task or interrupt code. A clean way to add conditional debugging statements to a C program is to use a MACRO such a D(bug)) by including lines like the following in your program. Set MYDEBUG to 1 to turn on debugging. Set bug to printf for printf debugging, or to kprintf (and link with debug.lib) for serial debugging, or to dprintf (and link with ddebug.lib) for parallel debugging. The D(bug()) macro is neater in your code because it can be indented and you need not surround it with any #ifdef directives yourself. Just be careful to remember to put two close parens at the end of each D(bug()) statement, before the semicolon.

```
/******      debug macros      *****/
#define MYDEBUG 1
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);
#define DEBTIME 0
#define bug printf
#if MYDEBUG
#define D(x) (x); if(DEBTIME>0) Delay(DEBTIME);
```

```
#else
#define D(x) ;
#endif /* MYDEBUG */
/***** end of debug macros *****/
```

Example macro usage:

```
win = OpenWindow(&mynewwin);
D(bug("Opened window at %lx", win));
```

Note that `kprintf()` and `dprintf()` debugging can be used inside even the lowest level task or interrupt code (although you better keep output down to a minimum during interrupts!). The `DEBTIME` (Delay) in the macro above must be 0 however if you are doing output from anything other than a Process.

Finding the Cause of Enforcer and Mungwall Hits

By using Enforcer and Mungwall while you are developing your software, you can catch problems as soon as they are introduced and greatly cut down your debugging time. It is especially useful to place conditional remote debugging statements in your code as you write each routine so that they can easily be turned on when a problem occurs. You will easily be able to pinpoint the problem area when the `kprintf()` (or `dprintf()`) output is intermixed with the Enforcer or Mungwall output. The remote debugging commands `kprintf()` and `dprintf()` are available in the linker libraries `debug.lib` (serial) and `ddebug.lib` (parallel) respectively. These linker libs are supplied with some compilers and are also available on Commodore's Native Developer Update disks.

Some people prefer to use a source-level or single-stepping debugger in combination with Enforcer and Mungwall when tracking down a problem to single-step through their code until the hit occurs. A different low-level method of locating Enforcer or Mungwall hits is to disassemble program memory where the hit occurred. If the hit occurred in ROM, first try using `OWNER` to determine ROM subsystem of that address. For example, `OWNER 0x202442` might return the following on a soft-kicked A2500:

<u>Address</u>	<u>Owner</u>
00202442	in resident module: exec 39.47 (28.8.92)

Then use `LVO` to determine the probable function at that address within the subsystem:
`LVO exec romaddress=0x202442`

Closest to \$202442 without going over: `exec.library LVO $fe0e -498`
`OpenResource()` jumps to \$202358 on this system

(See the Owner and LVO section for more information on these tools)

If this does not give enough clue, try disassembling just before other ROM and RAM addresses shown in the Enforcer stack dump line. If code is found, these may be application or ROM functions which called the routine that generated the hit.

If the hit occurs in RAM code, use various methods to match up the disassembly with your own code. Assembly programmers can just compare the disassembly to their source. Others may wish to take the hex values of a sequence of position-independent 68000 instructions near the hit (i.e., no addresses except for offsets and branches) and do a search for this binary pattern in your object modules. If you find the pattern, do a mixed source and object disassembly of that object module (for example, with SAS's OMD you could OMD >ram:dump mymodule.o mymodule.c) and then look in the output for instructions matching those where the hit occurred. Note that when a hit occurs in a disk-loaded device or library, it is currently not possible to determine who owns the code where the hit occurred. This is because it is up to the device or library to store its seglist wherever it sees fit in its own library or device base. If you suspect that your hit is occurring in a disk-loaded library or device, you can try searching suspected disk libraries or devices for a match of the hex pattern of position-independent code near the hit.

Deciphering Enforcer and Mungwall Output

Enforcer and Mungwall provide lots of valuable information to help debug your hits.

Sample Enforcer Output:

Here is a sample Enforcer hit caused by a program called *lawbreaker*.

```
WORD-READ from 00000014                      PC: 075899A6
USP: 075A8BE0 SR: 0015 SW: 0769 (U0)(F)(D) TCB: 075AE468
Data: DDDD0041 DDDD1100 DDDD2200 DDDD3300 DDDD4400 DDDD5500 DDDD6600 DDDD7700
Addr: AAAA0000 AAAA1100 AAAA2200 AAAA3300 AAAA4400 AAAA5500 07400810 -----
Stck: 00000009 00000008 00000007 00000006 00000005 00000004 00000003 00000002
Stck: 00000001 00F92A18 00001000 075AEE4C BBBB BBBB BBBB BBBB
Name: "Shell Process" CLI: "lawbreaker" Hunk 0000 Offset 00000096
```

```
LONG-WRITE to FEEDFACE                      data=DDDD0041 PC: 075899B8
USP: 075A8BE0 SR: 0018 SW: 0709 (U0)(F)(-) TCB: 075AE468
Data: DDDD0041 DDDD1100 DDDD2200 DDDD3300 DDDD4400 DDDD5500 DDDD6600 DDDD7700
Addr: AAAA0000 AAAA1100 AAAA2200 AAAA3300 AAAA4400 AAAA5500 07400810 -----
Stck: 00000009 00000008 00000007 00000006 00000005 00000004 00000003 00000002
Stck: 00000001 00F92A18 00001000 075AEE4C BBBB BBBB BBBB BBBB
Name: "Shell Process" CLI: "lawbreaker" Hunk 0000 Offset 000000A8
```

Here is an explanation of the some of the important information:

PC:

Program Counter. This is the address in memory where the program was executing instructions when the hit occurred. For some kinds of hits this will often be the address of the instruction after the hit. Note that if your program passes a bad pointer or an improperly initialized structure to a system ROM routine, you may cause ROM code to read or write to an illegal address.

TYPE-SIZE from or to

(e.g., **WORD-READ from 00000014**):

This is the type of illegal access and the address where it occurred. In the first example, the illegal access occurred at address \$14, and is specified as a WORD-READ access. This means the illegal memory access was an attempt to read a word (2 bytes) at address \$14. Low memory accesses are often caused by NULL pointers to structures. If, for example, your code or a ROM routine references a structure member at offset \$14, and what you provided or used a NULL structure pointer, Enforcer will pick up a hit at address \$14. Other illegal READ accesses are BYTE-READ (generally caused by a bad string pointer), and LONG-READ (generally caused by a bad pointer or a bad pointer within a structure). An Enforcer hit will occur whenever a program attempts to read low memory addresses (generally caused by a NULL pointer of some type) or non-existent memory addresses (generally caused by an uninitialized pointer). When Enforcer catches an illegal READ access, it reports the access and prevents the program from reading the address by returning data of zero.

On illegal WRITE accesses (i.e., attempts to write to ROM, low memory, or non-existent memory) Enforcer will report BYTE-WRITE, WORD-WRITE, or LONG-WRITE hits (such as in the second example). These mean the access would have illegally and dangerously written to memory which should not be written to, quite possibly causing memory to be trashed. WRITE hits can be caused by bad pointers or bad code. Fortunately, Enforcer prevents the actual memory trashing from occurring, and instead just reports it. This can allow debugging severe low memory trashing problems which would normally crash your machine. In the second example above, the CLI program *lawbreaker* tried to write the long value \$DDDD0041 to the address \$FEEDFACE.

Occasionally you will see an INSTRUCTION access of illegal memory meaning the PC (Program Counter) is at an address it has no legal reason to be at. Generally this is caused by trashed code, a trashed return address on your stack, or an invalid library base. Hint - when an INSTRUCTION Enforcer hit occurs at a negative address, it is most often caused by a

JSR to an LVO (negative word offset) from a NULL library base. Check the Stack dump lines for a possible return address to the code that made the bad library call.

"data=xxxxxxx":

On WRITE hits, Enforcer will show you the data that would have been written to the illegal address.

Register Dump Lines:

All of the middle Enforcer lines show the contents of the program's registers and stack at the time of the hit. The Data: line shows the D registers (D0 though D7). The Addr: line shows the A registers (A0 though A6). A7 (the Stack register) is shown as USP, and the contents of the top of the stack is shown in the Stck: lines (note - more Stck: lines are available via a command line option of the new Enforcer). The USP line also contains the contents of additional processor status registers, and also the address of the Task Control Block (TCB). On the same line are symbols which specify whether a Forbid (F) and/or a Disable (D) was in effect when the hit occurred.

Name and Hunk Offset:

Example: Name: "Shell Process" CLI: "lawbreaker" Hunk 0000 Offset 000000A8

This line shows the task name, and CLI command name (if any) of the task or command that was executing when the hit occurred. If possible, Enforcer also provides a hunk offset to where the program counter was if the hit occurred within the program's own code instructions.

Sample Mungwall Output

Here are sample Mungwall hits by a program called *mungwalltest*. Additional explanation has been added in parentheses below each hit. For reference, the arguments for memory functions are AllocMem(size,type) and FreeMem(address,size). The A: and C: addresses are Mungwall's guess at the address from which AllocMem was called. The A: address is the address if the caller was assembler code. The C: address is the probable address if the caller was C code, assuming a standard stub. Since Mungwall is wedged into the memory allocation functions, it can only guess the caller's address based on what is pushed on the stack. The "task" address on the first line of a Mungwall hit is the task address of the caller. Mungwall checks for incorrect allocations (such as 0 size) during AllocMem, and checks for incorrect deallocations or trashing around allocations during FreeMem. If trashing or incorrect deallocation is detected, Mungwall will both report it and refuse to free the memory.

Note that Mungwall has special code to prevent trapping the partial (wrong size) deallocations that are performed by some version of layers.library. If any other debugging tools are also wedged into AllocMem and FreeMem, Mungwall's A: and C: addresses may be thrown off by additional information pushed on the stack, and Mungwall will also be unable to screen out any partial layers deallocations (which will show up as hits on your task's context).

AllocMem(0x0,10000) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB12 C:0x3EF552 SP:0x4559FC
(means mungwalltest tried to allocate 0 bytes of memory)

FreeMem(0x0,16) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF598 SP:0x4559F4
(mungwalltest tried to free memory it never got - i.e., at address 0)

FreeMem(0x2FE7C0,0) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF5A8 SP:0x4559EC
(mungwalltest tried to free 0 bytes of memory)

Mis-aligned FreeMem(0x2FE7C4,16) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF5B6 SP:0x4559E4
(mungwalltest tried to free memory at an incorrect address)

Mismatched FreeMem size 14!
Original allocation: 16 bytes from A:0x3EBB12 C:0x3EF574 Task 0x3E08F0
Testing with original size.
(mungwalltest tried to free a different size from what it allocated)

19 byte(s) before allocation at 0x2FE7C0, size 16 were hit!
>\$: BBBB BBBB BB536572 6765616E 74277320 50657070 65722000
(memory before this allocation was trashed; trash shown at right)

8 byte(s) after allocation at 0x2FE7C0, size 16 were hit!
>\$: 75622042 616E6400 BBBB BBBB BBBB BBBB BBBB BBBB
(memory after this allocation was trashed; trash shown at left)

FreeMem(0x2FE7C0,14) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF5F4 SP:0x4559D0
(deallocation refused due to trashing explained in last two reports)

Failed memory allocation!
AllocMem(0x50000000,1) attempted by 'flush' (task 0x3E08F0)
from A:0x453E96 C:0x1 SP:0x4820F4
(SHOWFAIL option shows flush tried to allocate memory and failed)

As you can see, Mungwall alone can catch a large variety of memory-related software problems. But one of the most important benefits of Mungwall is that by filling freed memory with nasty 32-bit values, it can force subtle memory misuse problems into the open, by often causing the misuses to access addresses that can be trapped on by Enforcer.

Enforcer and Mungwall are required tools for the development of bug-free Amiga software. If you don't have an MMU, get one. The investment in an A3000, 68030 card, or 68020+MMU card will quickly pay for itself by cutting down on your development and allowing you to catch and find software problems with Enforcer. Enforcer and Mungwall are not just for developers and QA departments. Anyone who uses or reviews in-house software or software for purchase or contract can benefit your company by catching hidden software problems during normal usage and examination of the programs. Many people at Commodore and other companies run Enforcer and Mungwall all of the time. Keep that in mind if you are trying to impress a company with your software!

Other Debugging and Testing Tools

The next three tools, Memoration and Scratch by Bill Hawes, and IO_Torture by Bryce Nesbitt are important QA tools for testing the robustness of all application failure code, for uncovering illegal register usage in assembler code, and for detecting unsafe use of device I/O requests. Following notes on these three are notes on some of the additional debugging and stresstest tools provided by CATS.

Memoration

Memoration by Bill Hawes is a tool to selectively limit the ability of a task or module to allocate memory, thereby simulating the effects of a low-memory condition. It provides the unique ability to selectively cause each individual memory allocation of a program, whether direct or indirect, to fail, thereby allowing you to test the failure path and abort code at every point in your application code.

Memoration works by SetFunctioning the Exec AllocMem() and/or AllocVec() entries and then screening the requests. If a request from a particular task or range of addresses is received, memoration returns a zero instead of passing it through to AllocMem().

When a task or module is denied a memory request, memoration sends a message to the serial port identifying the client task ID, the address it was called from, and the size of the denied request. If the software can't handle being denied its memory request, this message will typically be followed by a series of enforcer reports telling of how the software went ahead and wrote to location 0.

Command-Line Parameters.

Memoration accepts command-line parameters to specify the module or task name and the range of memory sizes to disallow. The argument template is

MODULE, TASK/K, CLI/K/N, OFF/S, MIN/K/N, MAX/K/N, AFTER/K/N,
EVERY/K/N, ALLADDR/S, ALLOCVEC/S, CHIP/S, FAST/S, BACKTRACE/K/N

and the specifications can be changed at any time by reissuing the command.

MODULE is the name of a ROMTag or library.

The resident modules are searched first, followed by a search of the system library list. When an entry is found, the range of addresses encompassing its code is determined using several methods. For ROMTags the range extends from the ROMTag itself to the next higher module, or to RT_ENDSKIP if no higher module exists. For libraries a certain amount of voodoo is required, as the location of the library ROMtag isn't stored in the (public) library structure. In this case memoration examines the LVOs to determine the lowest and highest addresses, and then searches for a ROMtag in the range (low-\$2000,high+\$2000). If a ROMTag is found, memoration uses the smaller of the ROMTag address and the lowest LVO address as the low limit, and the larger of the RT_ENDSKIP address and the highest LVO address as the high limit.

TASK specifies the name of a task to trap.

The task must exist at the time memoration is run, and for best results should persist for the course of testing. If you're using WShell (as you should be) you can define a name for a particular shell instance by using "newwsh name sucker".

CLI specifies a shell number as the task to trap.

MIN specifies the minimum memory request to trap.

The default is 0.

MAX specifies the maximum allocation to trap.

The default is 2000000.

OFF turns off memory trapping.

The code patch is left intact, but won't trap any requests until enabled again. AllocMem() and AllocVec() traps can be turned on and off separately.

ALLOCVEC sets the trap for the AllocVec() entry, instead of AllocMem(). Both functions can be trapped independently.

AFTER specifies the number of allocations (within other specifications) to pass before beginning the trap.

EVERY traps every Nth allocation meeting the specifications.

ALLADDR sets the address range to all memory.

CHIP limits the trap to Chip memory specifications.

FAST limits the trap to Fast memory specifications.

BACKTRACE specifies the number of longwords of stack backtrace desired.

Examples:

```
memoration myprog after 15 ; after .15th allocation, deny myprog any memory
memoration dos.library ; disable all DOS allocations
memoration task DFO min 400 ; disable larger allocations by DFO:
memoration icon.library task Workbench every 3
memoration console.device min 40 backtrace 8
```

Example test session to test failure of every allocation an application makes:

Open two shells - one for memoration (denoted by 1> below) and one for myprog (denoted by 2> below). Alternately, you could start myprog from Workbench. Run Enforcer and Mungwall. Have serial or Sushi debugging set up.

```
1> memoration myprog after 0
2> myprog ; a failure occurs but hopefully no hits or crashes
1> memoration off (exit myprog if it made it up)
```

```
1> memoration myprog after 1
2> myprog ; a failure occurs but hopefully no hits or crashes
1> memoration off (exit myprog if it made it up)
```

```
1> memoration myprog after 2
2> myprog ; a failure occurs but hopefully no hits or crashes
1> memoration off (exit myprog if it made it up)
```

Note - until you get up to a higher "after" number, myprog will likely fail during the startup code it is linked with - i.e., before even reaching your main() entry point.

Further Notes. Memoration uses a seglist-split for its code patch, and so shouldn't be made resident, at least not on the first execution. Memoration was written by William S. Hawes.

Scratch

Scratch by Bill Hawes purposely trashes the scratch registers (D1, A0, A1) on exit from library functions of any "scratched" library. This will point out assembler callers that accidentally or purposely depend upon getting "useful" values from those registers. It is extremely important that ALL assembler applications be tested with Scratch. Even minor changes to the OS can change the values that happen to be left over in registers on exit from a system library function. Assembler code that is accidentally reusing a scratch register (for example A1) after a system call, or code that is looking at the wrong register for a result has the potential to break badly with even a minor change to the OS. Scratch can catch these problems before you ship. To use Scratch, see the script *SCRATCHALL.SCRIPT* which properly excepts certain system functions from scratching.

IO_Torture

IO_torture is a tool which can be used to check for improper use or reuse of device I/O requests.

IO_torture should be part of the standard test suite for all products.

Exec device I/O usage is tracked by IO_torture. If an IORequest is reused while still active, IO_torture will print a warning message on the serial port (parallel for IO_torture.par).

The current plan of IO_torture includes:

- ☐ SendIO()

Check that message is free. Check for ReplyPort. Be sure request is not linked into a list.

☐ **BeginIO()**

Check that message is free. Check for ReplyPort. Be sure request is not linked into a list.

☐ **OpenDevice()**

Mark message as free. If error, trash IO_DEVICE, IO_UNIT and LN_TYPE. Be sure request is not linked into a list.

☐ **CloseDevice()**

Check that message is free. Trash IO_DEVICE, IO_UNIT.

IO_torture does not currently check for another common mistake: After virtually all uses of AbortIO(IORquest), there should be a call to WaitIO(IORquest). AbortIO() asks the device to finish the I/O as soon as possible; this may or may not happen instantly. AbortIO() does not wait for or remove the replied message.

Note regarding NT_MESSAGE: NT_MESSAGE would seem to be the correct node type for an I/O request which is newly initialized. However, part of how IO_torture works is that it makes sure in-use requests are marked as NT_MESSAGE, and would normally complain if such an NT_MESSAGE came through BeginIO() or SendIO() (as it would signify reuse before ReplyMsg). So that IO_torture will not complain about a freshly initialized MT_MESSAGE I/O request, IO_torture also checks to see if the message has ever been linked into a list. If it has not, IO_torture will let the NT_MESSAAHE marked request by without complaining. This is necessary because the amiga.lib CreateExtIO() and CreateStdIO() historically set a newly created I/O request node type to NT_MESSAGE.

Devmon

Devmon is a tool which allows you to monitor the activity of any exec device. It is extremely useful for debugging both application use of a device and your own device drivers. Devmon captures (or outputs to serial or Sushi) debugging information every time any vector of a monitored device is entered. Optionally, Devmon can also report on the entry to (and in some cases, exit from) the exec library functions which call the device.

Usage:

devmon name.device unitnum [remote] [hex] [allunits] [full]

remote

means serial debugging output

hex

means show values in hex

allunits

monitor regardless of unit pointer (required if device gives a new unit pointer to every opener)

full

means also monitor exec device-related library functions

Some sample Devmon output as generated by:

```
devmon clipboard.device 0 allunits hex full remote
```

In the following regular devmon output lines, **UPPERCASE**: signifies entry to an actual device vector, while **MixedCase**: signifies entry to (or Rts from) an exec function. The number preceded by "@" is the address of the I/O request. The single letters following are abbreviations for the fields of an **IOStdRequest** (C means **io_Command**, F means **io_Flags**, E means **io_Error**, etc.). When Devmon is exited, it outputs a key to these fields.

Here we see C:ConClip executing **DoIO()** of a request containing **CMD_WRITE** (C = 3, as defined in **exec/io.h**) of 8 bytes (L=\$8) of data at \$30BDF0. We see the same I/O request make it to the **BEGINIO** vector of the device, and then we see the **DoIO()** completing successfully with an **io_Actual** (A=) of 8.

```
DoIO : @5478294 C= 3 F=$81 E=0 A=$4 L=$8 D=$30BDF0 O=$0 (C:ConClip)
BEGIN: @5478294 C= 3 F=$81 E=0 A=$4 L=$8 D=$30BDF0 O=$0 (C:ConClip)
DoRts: @5478294 C= 3 F=$81 E=0 A=$8 L=$8 D=$30BDF0 O=$8 (C:ConClip)
```

TNT

TNT installs a trap handler that replaces Software Error requesters with a large requester containing informative debugging information. TNT can be called in your user-startup (it does not need to be run). But it does not get along well with trap-based single-stepping debuggers. So if you plan to run a debugger, turn off TNT with **TNT OFF**.

TNT requesters contain PC, task/command name, SP, SSP, register, and stack contents information similar to that displayed by Enforcer.

Owner and LVO

Owner and LVO are very useful for determining the owner of a memory address. This can help you determine the location and even the cause of an Enforcer or Mungwall hit.

LVO requires the Amiga FD files for your OS version in a directory assigned the logical name FD:. The FD files are provided with the includes and linker libs for the OS.

When you get a hit, use OWNER to determine if the address is in a ROM module (such as exec or intuition), or in the loaded code, stack, port, or other memory owned by a program. Note that owner can not determine if an address is in the code of a disk-loaded device or library because there is no standard place for devices and libraries to store their seglist.

Examples:

```
1> owner 0x202443
```

<u>Address</u>	<u>Owner</u>
00202442	in resident module: exec 39.47 (28.8.92)

Now use LVO to determine the probable function at that address within the subsystem:

```
1> LVO exec romaddress=0x202442
```

Closest to \$202442 without going over: exec.library LVO \$fe0e -498
OpenResource() jumps to \$202358 on this system

LVO can also be used as a programming helper to list one function and its FD comment:

```
1> LVO exec AllocMem exec.library LVO $ff3a -198 AllocMem()  
AllocMem(byteSize, requirements) (d0/d1)
```

LVO will list all of a library's LVOs if no function name is provided. With the optional CONTAINS keyword, LVO will also list the addresses each function jumps to on YOUR system (different on different systems). Note that if you have debugging tools installed which use SetFunction (for example, Mungwall, Devmon, or Scratch), LVO will not be able to determine the ROM address of the SetFunctioned library functions because the LVOs of those functions will point to the SetFunctioned RAM code.

LVO can also create command lines for the Wedge program.

Note that LVO and Wedge can only interact with actual library functions, not for their various stub interfaces which only exist in linker libraries. Actual library functions are the functions listed in the FD file for the libraries. Note also that under new versions of the OS, some older library functions become unused by the OS and newer functions are used in their place.

Wedge

Wedge is a tool for wedging into almost any system library function and monitoring calls to and results from the function, for any or all system tasks. Sometimes it can be more efficient to quickly wedge and monitor a system function rather than add debugging code and recompile. It used to be very difficult to create command line arguments for wedge. But LVO can generate a template command line to "wedge" into any system function.

Example: Creates a wedge command for wedging into OpenScreen

```
1> LVO intuition CloseScreen wedgeline run wedge intuition
0xffbe 0x8100 0x8100 opt r "c=CloseScreen(screen) (a0) "
```

If you execute the WEDGE command line above, you will receive serial reports on every call to CloseScreen. To turn off the wedge, type WEDGE KILLALL. See Wedge.doc for more information.

Tstat

Tstat is a handy little tool for checking the signals, priority, state, and other Task control block variables of any task or command. Tstat also does its best to show the task registers as they were saved at task-switch time, and the stack usage of the task or command. Note however that Tstat is looking at private exec Task context information and therefore often needs to be updated for each OS release, and may misinterpret the task context data in unusual conditions such as a task switch which occurs in the middle of an FPU instruction. But it is very useful when checking for lost signals, bad Forbid or Disable counts, and hung Waits. Tstat can monitor once, or periodically. For example, `tstat MyProg -4` would monitor MyProg every 4/50ths of a second until Ctrl-C is hit. In a pinch, Tstat can be used as a poor man's logic state analyzer to track another program stuck in a loop.

Other Memory Tools

EatMem

is an interactive tool for scaling back the apparent amount of memory available to the system. It is useful for testing applications in a simulated low-memory situation, as well as for testing how an application deals with only chip memory or only fast memory. EatMem requires at least V37 OS.

MemList

displays all memory blocks (both free and in-use) in the system. It can be useful for debugging fragmentation/deallocation problems. See also the NAMETAG option of Mungwall, and Munglist.

Frag

displays a chart of current system memory fragmentation.

Memmon

saves current free memory and a comment to a file on demand. It is useful for documenting memory usage while testing various program operations.

Flush

does two large allocations designed to fail and thereby cause all disk-loaded devices, libraries, and fonts which are not currently in use to be flushed from the system. Useful when doing memory-loss testing and device or library development.

Snoop

can be used to snoop memory allocations, but has largely been replaced by the more flexible Mungwall SNOOP option. SnoopStrip is used to discard allocation/deallocation pairs from captured Snoop or Mungwall SNOOP output.

Report

All bug reports, compatibility problem reports, and enhancement requests must be generated with the latest *Amiga Report* program (currently version 39.6), or must be compatible with the output of the *Amiga Report* program. This makes automatic submission and routing of bug reports possible. The *Report* program is included on most developer tool and DevCon disks. *Report HELP* will output instructions and submissions addresses.

We strongly request that you submit your reports electronically if possible. Please use report. Only mail them on paper if you have no other method available.

European ADSP users

Post in appropriate closed adsp bugs topic

BIX/CIX

Post bugs in the appropriate bugs topic of your closed conference.

UUCP

to uunet!cbmvax!bugs OR rutgers!cbmvax!bugs OR bugs@commodore.COM
(enhancement requests to cbmvax!suggestions instead of cbmvax!bugs)

Mail

Mail individual bug reports in Report format, on disk if possible.

European developers:

Mail bug reports to your support manager unless your support manager says to mail directly to West Chester.

U.S./others mail to:

ATTN: BUG REPORTS
Amiga Software Engineering,
CBM
1200 Wilson Drive
West Chester, PA., 19380, USA

Please make sure the initial one-line bug description you provide in each of your reports is as explicit as possible. Engineering's bug summary reports and bug processing tools often list only the one-line description for each bug. "Bug in intuition" is a useless title for an intuition subsystem bug. "Pixel trash when window dragged left" is a much better title.

The latest Report program always includes a list of the currently acceptable subsystems for bug reports and enhancement requests. These allow reports to be properly routed. The current list is:

A2024	A2065	A2090.A2090A	A2091.A590
A2232	A2300	A2410	A2620
A2630	A3000	AmigaBasic	aa.chips
alink	amiga.lib	amigaguide	amigaterm
amigavision	appshell	arexx	art
as225	asl.library	audio.device	autoconfig
battclock	battmem	bootmenu	bridgeboard
bru	bullet	cdos.command	cdtv
cia.resource	clipboard	commodities	con-handler
console.device	creditcard	crossdos	custom.chips
datatypes	debug.lib	disk.resource	diskfont
documentation	dos.library	exec	expansion
filesysres	filesystem	fonts	fountain
gadget.classes	gadtools	gameport	genlock
graphics	hardware	hdbackup	hdtoolbox
icon.library	iconedit	ide.device	iffparse
input.device	installer	intuition	iprefs
keyboard	keymap	keymaps	kickmenu
layers	locale.library	mathffp	mathieee
mathieedoub	mathieeesing	microemacs	monitors
multiview	narrator.device	new.look	parallel.device
port-handler	potgo.resource	preferences	printer.device
printer.driver	queue-handler	ram-handler	ramdrive.device
ramlib	resource	sana2	script
scsi.device	serial.device	setpatch	shell
speak-handler	startup	strap	system.command
toolkit	toolmaker	tools	trackdisk
translations	translator	unix	util.command
utility.library	wack	wbttag	workbench



1

2

3

The book has been kindly donated for scan by Bo Zimmerman (<http://www.zimmers.net/cbmpics>).

Hardware/software used to digitize this tome:

- Fujitsu ScanSnap S1300i
- Adobe Acrobat XI Pro
- GIMP

For any suggestions, contact: jman@storiepvtride.it

– jman
20150531

The book has been kindly donated for scan by Bo Zimmerman (<http://www.zimmers.net/cbmpics>).

Hardware/software used to digitize this tome:

- Fujitsu ScanSnap S1300i
- Adobe Acrobat XI Pro
- GIMP

For any suggestions, contact: jman@storiepvtride.it

– jman
20150531